cf/x

DML

(Dynamic Mission Library)

for DCS

Version II

© 2022 - 2024 by Christian Franz and cf/x

Version 2.3.7 – 2024-11-21

Table of Contents

1 Foreword to Version 2 20
1.1 How to get started quickly 20
1.2 What you need to know already 20
1.3 No Scripting, no Lua 21
1.4 DEMOS! 21
2 Preface 22
2.1 What is DML? 22
2.2 DML in a Nutshell 24
2.3 About this Document 27
2.4 DISCLAIMER 28
2.5 Copyright 28
2.6 Acknowledgements 29
3 Introduction to DML 31
3.1 DML Modules 32
3.1.1 “Building Blocks” 33
3.1.2 “Big Ticket” Modules 36
3.1.3 Server Modules 38
3.2 Other Attractions (Overview) 38
3.2.1 The Debugger 38
3.2.2 Persistence 38
3.2.3 Multi-Player support built in 39
3.2.4 Fully Documented Demos 39
4 Using DML in your missions 41
4.1 The Basics 41
4.1.1 Installing DML 41
4.1.2 Add DML to your Mission 41
4.1.3 Adding Building Blocks to your mission 45
4.1.4 Placing DML Modules on the map 45
4.1.5 Wait… Attribute Editor? What’s that? 46
4.1.6 Attributes and their Defaults 47
4.1.7 That text in the upper right corner… 47
4.1.8 Adding more attributes 48
4.1.9 The Yeller Model (Simplification) 50
4.1.10 The Flags model 54
4.1.11 What? Where? When? How? 58

4.1.12 So what? 59
4.2 DML “Crown Jewels” 61
4.2.1 cloneZones 61
4.2.2 messenger 61
4.2.3 radioMenu 62
4.2.4 heloTroops 62
4.3 DML “Dark Horses” 63
4.3.1 Persistence 63
4.3.2 ssbClient / slotty 63
4.3.3 THE DEBUGGER 63
5 DML Reference 65
5.1 DML Core Concepts 65
5.1.1 Zones and Attributes 65
5.1.2 So what, exactly, are Attributes? 66
5.1.3 Module Configuration Zones 67
5.1.4 Zone Shape 68
5.1.5 Flags in DML 69
5.1.6 Further thoughts 77
5.1.7 Multiple Output Flags 80
5.1.8 Attribute Synonyms – Why? 80
5.1.9 DML Flags: Named, and Local Flags 81
5.2 Special Concepts 84
5.2.1 Orders (Spawn Zones) 84
5.2.2 Spawn Formations (spawners) 86
5.2.3 Spawning: Type String and Type String Arrays 87
5.2.4 Zone Ownership / Owned Zones 87
5.3 “Building Blocks” Modules 89
5.3.1 Core Abilities (shared by all DML Modules) 90
5.3.2 smokeZones 96
5.3.3 rndFlags 99
5.3.4 pulseFlags 106
5.3.5 delayFlags (“Timer”) 109
5.3.6 raise Flag 112
5.3.7 xFlags (Flag Combining / Testing) 114
5.3.8 changer 121
5.3.9 countDown 125
5.3.10 objectDestructDetector (map/scenery object destruction) 129

5.3.11 SpawnZones 132
5.3.12 clone Zones 139
5.3.13 objectSpawnZone 166
5.3.14 cargoReceiverZone 172
5.3.15 artilleryZones 174
5.3.16 Artillery UI 177
5.3.17 ownedZones 180
5.3.18 factoryZones 192
5.3.19 FARP Zones 200
5.3.20 mapMarkers 204
5.3.21 NDB 205
5.3.22 messenger 209
5.3.23 unit Zone 221
5.3.24 groupTracker 226
5.3.25 wiper (Unit/Object/Debris removal) 232
5.3.26 radioTrigger 237
5.3.27 baseCaptured 239
5.3.28 radioMenu 241
5.3.29 delicates 252
5.3.30 Impostors 255
5.3.31 persistence 259
5.3.32 unitPersistence 263
5.3.33 sequencer 265
5.3.34 fireFX 270
5.3.35 LZ (Landing Zone) 273
5.3.36 counter 276
5.3.37 valet (greet and goodbye players) 278
5.3.38 flareZone 284
5.3.39 playerZone 287
5.3.40 TACAN 289
5.3.41 (Simple) Mission Restart 293
5.3.42 airfield 294
5.3.43 ownAll 300
5.3.44 groundExplosion 304
5.3.45 Sweeper 308
5.3.46 twn (automatic Town locale bridge) 310
5.3.47 Convoy 311

5.3.48 WHpersistence 318
5.4 “Big Ticket” (“drop-in”) Modules 319
5.4.1 Player Score / PlayerScoreUI 319
5.4.2 cfxHeloTroops (Troop Transport) 334
5.4.3 csarManager 339
5.4.4 autoCSAR 347
5.4.5 Limited Airframes 349
5.4.6 Guardian Angel 353
5.4.7 Parashoo 358
5.4.8 Civ Air 359
5.4.9 Recon Mode 368
5.4.10 ssbClient 377
5.4.11 ssbSingleUse (tbc) 381
5.4.12 unGrief (enforce PVP/PVE mode, no friendly kills) 382
5.4.13 Willie Pete (FAC - Artillery for Targets marked with Smoke) 386
5.4.14 ASW (Anti-Submarine Warfare / Sub Hunters) 392
5.4.15 Taxi Police 402
5.4.16 Shallows (automatically remove ship husks) 406
5.4.17 StopGap and StopGapGUI 407
5.4.18 Sitting Ducks 412
5.4.19 NoGap and NoGapGUI 414
5.4.20 Bomb Range 418
5.4.21 TDZ (Touch-Down Zone) 425
5.4.22 Scribe (player logging) 430
5.4.23 Civ Helo 434
5.4.24 Slotty 438
5.4.25 Reaper 440
5.4.26 Inferno 447
5.4.27 Airtank 452
5.4.28 Module Name 456
5.5 Server-based modules 457
5.5.1 smrGUI 458
5.5.2 stopGapGUI 459
5.5.3 noGapGUI 460
5.5.4 Tickle (tcliGUI) 461
6 Persistence 468
6.1 What is persistence? 468

6.1.1 FOR YOUR OWN PROTECTION: READ THIS CAREFULLY 468
6.1.2 De-Sanitizing LFS and IO 469
6.1.3 Security Threat Assessment 470
6.1.4 Mitigating Measures (if you are uncomfortable with persistence) 471
6.2 Why persistence is currently difficult in DCS 471
6.3 Using Persistence in your Mission 473
6.4 Dedicated persistence modules 473
6.5 Using persistence with DML modules 473
6.5.1 Restarting from scratch: 'Fresh Start' 474
6.5.2 Restart after Mission Update: versionID attribute 474
6.6 Mission Loading: Sequence of Events 474
6.7 Shared Data between Missions 475
6.8 ME Integration 476
6.8.1 versionID 476
6.8.2 Saving Flag Values 476
6.9 Configuration 476
6.9.1 Save Location 476
6.9.2 'Unconfigured' persistence 479
6.9.3 Persistence config zone 479
7 The Debugger 482
7.1 Using The Debugger (Summary) 484
7.1.1 Adding The Debugger to your mission (DML) 484
7.1.2 Adding The Debugger STANDALONE to your mission 484
7.1.3 During design (Mission Editor) 484
7.1.4 In-Mission 485
7.1.5 Debugger commands (overview) 486
7.2 Interactive Use during Missions 496
7.2.1 Remembering it for you wholesale: -help and -? 497
7.2.2 Fundamental commands: -show, -set, -flip and -inc 497
7.2.3 Let the Debugger work for you: -observe and -o 498
7.2.4 Losing interest: -forget 499
7.2.5 Annotating Message History: -note 499
7.2.6 Clearing the Slate: -reset 500
7.2.7 Getting the BIG picture: -snap and -compare 500
7.2.8 Getting cross with you: details on zones and flags: -x 501
7.2.9 Finding bad 'unconnected' flag names / commands: -x ! 502
7.2.10 Bringing in the (big) Guns: -spawn 503

7.2.11 Getting rid of ‘em: -remove 505
7.2.12 Smoke On! -smoke 506
7.2.13 Boom Baby! - boom 506
7.3 Killer Features 507
7.3.1 Q, no longer anon: -q 507
7.3.2 Analyze this: -a 507
7.3.3 W is for WARNING: -w 507
7.4 Big Brother: Observers 509
7.4.1 Creating an observer: -new and new for 509
7.4.2 Adding flags to observers: -observe with 510
7.4.3 Changing an observer’s condition: -update to 510
7.4.4 Supported Observer Conditions 511
7.4.5 Show me what you got: -list 511
7.4.6 Show and tell: -show observername 512
7.4.7 Who’s zoomin’ who: -who 512
7.4.8 … and Forget Me Nots: -forget with 513
7.4.9 Oh, snap! -snap observername 513
7.4.10 The spotless mind: -drop observername 513
7.4.11 Main switch: -start and -stop 514
7.5 Saving the Debugging Log: -save [filename] 515
7.5.1 Using -save [filename] 515
7.5.2 How to enable -save 516
7.5.3 Do I have to add persistence? 516
7.6 Integration With Mission Editor (Optional) 517
7.6.1 Creating Observers in ME 517
7.6.2 Dedicated and Stacked Zones, tracking local flags 518
7.6.3 Activating Event Monitoring 518
7.6.4 Setting up Generic Stand-ins 519
7.7 Adding The Debugger to your mission 520
7.7.1 Stand-Alone (NO DML in your mission) 520
7.7.2 DML-enhanced Missions 520
7.7.3 ME Attributes 521
7.8 Bug Hunt - A Live demo 524
7.8.1 Please ignore me: Self Test 524
7.8.2 Observing and setting flags to debug your mission 524
7.8.3 Setting flags to skip/advance mission stages 526
7.8.4 Using Observers and snapshots to debug your mission 527

7.8.5 Setting up Observers in ME 529
7.8.6 Debugging local flags, flag concurrency 530
7.8.7 Now hear this: debugMsg and sayWhen observer attributes 532
7.8.8 Mission Discussion: What DML does in this demo 533
7.9 Debug events and More (.miz) 534
7.9.1 Starting the mission 534
7.9.2 In-Mission 535
8 Tutorial / Demo missions 540
8.1 Overview 540
8.2 Smoke'em! DML Intro.miz 546
8.2.1 Demonstration Goals 546
8.2.2 What To Explore 546
8.2.3 Discussion 547
8.3 Object Destruct Detection (ME Integration).miz 548
8.3.1 Demonstration Goals 548
8.3.2 What To Explore 548
8.3.3 Discussion 549
8.4 ADF and NDB Fun.miz 550
8.4.1 Demonstration Goals 550
8.4.2 What To Explore 550
8.4.3 Discussion 552
8.5 Artillery Zones Triggered.miz 553
8.5.1 Demonstration Goals 553
8.5.2 What To Explore 553
8.5.3 Discussion 555
8.6 ME Triggered Spawns.miz 556
8.6.1 Demonstration Goals 556
8.6.2 What To Explore 556
8.6.3 Discussion 557
8.7 Spawn Zones (training and lasing).miz 558
8.7.1 Demonstration Goals 558
8.7.2 What To Explore 558
8.7.3 Discussion 559
8.8 Random Glory / Random Death (rndFlag) 561
8.8.1 Demonstration Goals 561
8.8.2 What To Explore 561
8.8.3 Discussion 564

8.9 Pulsing Fun.miz (pulseFlags) 566
8.9.1 Demonstration Goals 566
8.9.2 What To Explore 566
8.9.3 Discussion 567
8.10 Attack of the CloneZ.miz (Clone Zones and Delay) 568
8.10.1 Demonstration Goals 568
8.10.2 What To Explore 568
8.10.3 Discussion 574
8.11 Flag Fun – LOCAL FLAGS (& raiseFlags) 576
8.11.1 Demonstration Goals 576
8.11.2 What To Explore 576
8.11.3 Discussion 578
8.12 Once, twice, three times a maybe.miz (Event Count Down) 579
8.12.1 Demonstration Goals: The ~~un~~ending Spawner 579
8.12.2 What To Explore 580
8.12.3 Discussion 581
8.13 Bottled Messages.miz (Messenger, timeDelay) 582
8.13.1 Demonstration Goals 582
8.13.2 What To Explore 582
8.13.3 Discussion 583
8.14 Follow Me! (unit zones & messenger) 585
8.14.1 Demonstration Goals 585
8.14.2 What To Explore 585
8.14.3 Discussion 587
8.15 Clone Relations (Advanced Topic) 589
8.15.1 Demonstration Goals 589
8.15.2 What To Explore 590
8.15.3 Discussion 592
8.16 Moving Spawners I & II (SpawnZone and linkedUnit, GroupTracker) 594
8.16.1 Demonstration Goals 594
8.16.2 What To Explore 594
8.16.3 Discussion 595
8.17 Helo Trooper.miz 597
8.17.1 Demonstration Goals 597
8.17.2 What To Explore 597
8.17.3 Discussion 599
8.18 Helo Cargo.miz – cargo spawn & receive 601

8.18.1 Demonstration Goals 601
8.18.2 What To Explore 601
8.18.3 Discussion 602
8.19 Artillery with UI.miz 604
8.19.1 Demonstration Goals 604
8.19.2 What To Explore 604
8.19.3 Discussion 606
8.20 Missile Evasion (Guardian Angel).miz 608
8.20.1 Demonstration Goals 608
8.20.2 What To Explore 608
8.20.3 Discussion 609
8.21 Guardian Angel Reloaded (Guardian Angel 3.x) tbc 610
8.21.1 Demonstration Goals 610
8.21.2 What To Explore 610
8.21.3 Discussion 611
8.22 Recon Mode.miz 612
8.22.1 Demonstration Goals 612
8.22.2 What To Explore 612
8.22.3 Discussion 613
8.23 Recon Mode – reloaded (Recon Mode 2.x) 615
8.23.1 Demonstration Goals 615
8.23.2 What To Explore 615
8.23.3 Discussion 619
8.24 Owned Zones ME Integration.miz 620
8.24.1 Legacy Warning 620
8.24.2 Demonstration Goals 620
8.24.3 What To Explore 620
8.24.4 Discussion 621
8.25 FARP and away.miz (tbc) 622
8.25.1 Demonstration Goals 622
8.25.2 What To Explore 622
8.25.3 Discussion 622
8.26 Keeping The Score: Player Score.miz 623
8.26.1 Demonstration Goals 623
8.26.2 What To Explore 623
8.26.3 Discussion 624
8.27 The Zonal Countdown (Local/Global Flag Demo) 625

8.27.1 Demonstration Goals 625
8.27.2 What To Explore 625
8.27.3 Discussion 627
8.28 Frog Men Training.miz 629
8.28.1 Demonstration Goals 629
8.28.2 What To Explore 629
8.28.3 Discussion 632
8.29 CSAR of Georgia.miz 633
8.29.1 Demonstration Goals 633
8.29.2 What To Explore 633
8.29.3 Discussion 636
8.30 Track This! (Group Tracker) 637
8.30.1 Demonstration Goals 637
8.30.2 What To Explore 637
8.30.3 Discussion 639
8.31 Watchflags Demo 640
8.31.1 Demonstration Goals 640
8.31.2 What To Explore 641
8.31.3 Discussion 644
8.32 Viper with a double youu (Wiper) 645
8.32.1 Demonstration Goals 645
8.32.2 What To Explore 645
8.32.3 Discussion 647
8.33 Radio Go Go (Radio Trigger) 649
8.33.1 Demonstration Goals 649
8.33.2 What To Explore 649
8.33.3 Discussion 650
8.34 xFlags – Field Day (Decisions, Flag Testing) 651
8.34.1 Demonstration Goals 651
8.34.2 What To Explore 651
8.34.3 Discussion 653
8.35 Virgin (Civ) Air / Air Caucasus II / One-Way Air (CivAir) 655
8.35.1 Demonstration Goals 655
8.35.2 What To Explore 655
8.35.3 Discussion 657
8.36 Count Bases Blue (baseCaptured, xFlags) 659
8.36.1 Demonstration Goals 659

8.36.2 What To Explore 659
8.36.3 Discussion 660
8.37 Pilots at their Limits (Limited Airframes) 662
8.37.1 Demonstration Goals 662
8.37.2 What To Explore 662
8.37.3 Discussion 664
8.38 Gate and Switch (Changer – Pulse & Gated Switch) 665
8.38.1 Demonstration Goals 665
8.38.2 What To Explore 665
8.38.3 Discussion 668
8.39 Good Grief (unGrief) 670
8.39.1 Demonstration Goals 670
8.39.2 What To Explore 670
8.39.3 Discussion 671
8.40 The Danger Zone (MP – PvP Zones in PvE) 673
8.40.1 Demonstration Goals 673
8.40.2 What To Explore 673
8.40.3 Discussion 674
8.41 Reinforcements a la Carte (Radio Menu, Cloner, Dynamic Reinforcements, Mission Restart) 675
8.41.1 Demonstration Goals 675
8.41.2 What To Explore 675
8.41.3 Discussion 678
8.42 Delicate Subjects ('brittle' exploding units) tbc 681
8.42.1 Demonstration Goals 681
8.42.2 What To Explore 681
8.42.3 Discussion 682
8.43 Forever-looping Spawns (Server-friendly endless respawns) 683
8.43.1 Demonstration Goals 683
8.43.2 What To Explore 683
8.43.3 Discussion 686
8.44 Impossible Impostors – Using Impostors 687
8.44.1 Demonstration Goals 687
8.44.2 What To Explore 687
8.44.3 Discussion 691
8.45 Being Persistent (load and save a mission) 692
8.45.1 Demonstration Goals 692
8.45.2 What To Explore 692

8.45.3 Discussion 694
8.46 Sequencing Fun (Sequencers) 696
8.46.1 Demonstration Goals 696
8.46.2 What To Explore 696
8.46.3 Discussion 699
8.47 Departures and Landings.miz (LZ) 700
8.47.1 Demonstration Goals 700
8.47.2 What To Explore 700
8.47.3 Discussion 703
8.48 Willie Nillie (Willie Pete / Player Score drop-in) 704
8.48.1 Demonstration Goals 704
8.48.2 What To Explore 704
8.48.3 Discussion 707
8.49 Feats and autoCSAR 708
8.49.1 Demonstration Goals 708
8.49.2 What To Explore 708
8.49.3 Discussion 709
8.50 Slot-Blocking and You (ssbClient) 710
8.50.1 Demonstration Goals 710
8.50.2 What To Explore 710
8.50.3 Discussion 712
8.51 BFM Combat Trainer (moving zone, cloner & useHeading) 713
8.51.1 Introduction / Demonstration Goals 713
8.51.2 What To Explore 714
8.51.3 Discussion 716
8.52 Formation Trainer (messenger, counter, moving zone) 718
8.52.1 Demonstration Goals 718
8.52.2 What To Explore 718
8.52.3 Discussion 724
8.53 Hope you guess my name (Cloner Name Schemes) 725
8.53.1 Demonstration Goals 725
8.53.2 What To Explore 726
8.53.3 Discussion 733
8.54 I say hello goodbye (Valet) 735
8.54.1 Demonstration Goals 735
8.54.2 What To Explore 735
8.54.3 Discussion 737

8.55 Davy Jones' Rocker (ASW) 739
8.55.1 Demonstration Goals 739
8.55.2 What To Explore 739
8.55.3 Discussion 744
8.56 Taxi Police (policing airfield speed limitations) 746
8.56.1 Demonstration Goals 746
8.56.2 What To Explore 746
8.56.3 Discussion 749
8.57 Big Score: MoreScore.miz and LaterScore.miz 750
8.57.1 Demonstration Goals 750
8.57.2 What To Explore in MoreScore: Feats 750
8.57.3 What To Explore in LaterScore: Deferred Scoring / Kill Zones 753
8.57.4 Discussion 757
8.58 Effects with a Flare.miz (FlareZone) 759
8.58.1 Demonstration Goals 759
8.58.2 What To Explore 759
8.58.3 Discussion 760
8.59 Players in the Zone.miz (PlayerZone) 761
8.59.1 Demonstration Goals 761
8.59.2 What To Explore 761
8.59.3 Discussion 761
8.60 Not too shallow at all.miz (Remove ship husks) 763
8.60.1 Demonstration Goals 763
8.60.2 What To Explore 763
8.61 No Gap, Nop Glory.miz (StopGap, StopGapGUI) 764
8.61.1 Demonstration Goals (Single/Multiplayer) 764
8.61.2 What To Explore 764
8.61.3 Discussion 765
8.62 No Gap No, Problem (noGap).miz -- tbc 766
8.62.1 Demonstration Goals 766
8.62.2 What To Explore 766
8.62.3 Discussion 766
8.63 My first Factory.miz (OwnedZones 2, Factory 2) 767
8.63.1 Backstory 767
8.63.2 Demonstration Goals 767
8.63.3 What To Explore 767
8.63.4 Discussion 771

8.64 Owned Zones and Factories.miz (legacy migration) 773
8.64.1 Backstory 773
8.64.2 Demonstration Goals 773
8.64.3 What To Explore 773
8.65 Caucasus Hangar.miz (StopGap) 775
8.65.1 Demonstration Goals 775
8.65.2 What To Explore 775
8.65.3 Discussion 777
8.66 Sitting Ducks in a Barrel.miz (SittingDucks) 779
8.66.1 Demonstration Goals 779
8.66.2 What To Explore 779
8.66.3 Discussion 781
8.67 Flag Score.miz (Player Score: coalition scoring via flags) 782
8.67.1 Demonstration Goals 782
8.67.2 What To Explore 782
8.67.3 Discussion 783
8.68 Take on TACAN.miz (TACAN) 784
8.68.1 Demonstration Goals 784
8.68.2 What To Explore 784
8.68.3 Discussion 787
8.69 Civ Air International (Civ Air 2.0 “Off-map” locations) 788
8.69.1 Demonstration Goals 788
8.69.2 What To Explore 788
8.69.3 Discussion 789
8.70 Airbase mine.miz (airfield) 790
8.70.1 Demonstration Goals 790
8.70.2 What To Explore 790
8.70.3 Discussion 795
8.71 My Immortal (StopGap ‘refresh’ option) 796
8.71.1 Demonstration Goals 796
8.71.2 What To Explore 796
8.71.3 Discussion 797
8.72 Send in the Clones (requestable, airfield) 798
8.72.1 Demonstration Goals 798
8.72.2 What To Explore 798
8.72.3 Discussion 801
8.73 All is what I own.miz (ownAll) 803

8.73.1 Demonstration Goals 803
8.73.2 What To Explore 803
8.73.3 Discussion 806
8.74 Bombs Away.miz (bombRange) 808
8.74.1 Demonstration Goals 808
8.74.2 What To Explore 808
8.74.3 Discussion 811
8.75 Types and civil liveries (civAir liveries and type control) 813
8.75.1 Demonstration Goals 813
8.75.2 What To Explore 813
8.75.3 Discussion 814
8.76 Boom Boom (explosion, map object destruction revisited) 815
8.76.1 Demonstration Goals 815
8.76.2 What To Explore 815
8.76.3 Discussion 818
8.77 Landing Lessons (TDZ) 819
8.77.1 Demonstration Goals 819
8.77.2 What To Explore 819
8.77.3 Discussion 821
8.78 Player Score to Win (playerScore) 822
8.78.1 Demonstration Goals 822
8.78.2 What To Explore 822
8.78.3 Discussion 823
8.79 Clone Factory (CloneZone with Factory) 824
8.79.1 Demonstration Goals 824
8.79.2 What To Explore 824
8.79.3 Discussion 826
8.80 Airfield Mine (airfield, cloneZone, messenger) 828
8.80.1 Demonstration Goals 828
8.80.2 What To Explore 828
8.80.3 Discussion 830
8.81 On The Record (Scribe, Scribe + csarManager) 831
8.81.1 Demonstration Goals 831
8.81.2 What To Explore 831
8.81.3 Discussion 834
8.82 Civvy Heli 836
8.82.1 Demonstration Goals 836

8.82.2 What To Explore 836
8.82.3 Discussion 837
8.83 Red vs. Blue / Blue vs. Red (shared data) 838
8.83.1 Demonstration Goals 838
8.83.2 What To Explore 838
8.84 Non-SSB and SP slot blocking 841
8.84.1 Demonstration Goals 841
8.84.2 What To Explore 841
8.84.3 Discussion 842
8.85 What's on the cascading menu 843
8.85.1 Demonstration Goals 843
8.85.2 What To Explore 843
8.85.3 Discussion 844
8.86 Reaper, man 846
8.86.1 Demonstration Goals 846
8.86.2 What To Explore 846
8.86.3 Discussion 848
8.87 Escort a convoy to town (Convoy & twn) 850
8.87.1 Demonstration Goals 850
8.87.2 What To Explore 850
8.87.3 Discussion 853
8.88 The Yeller Model 855
8.88.1 Demonstration Goals 855
8.88.2 What To Explore 855
8.88.3 Discussion 855
8.89 Inferno at Sea (heloTroops: dropZone) 857
8.89.1 Demonstration Goals 857
8.89.2 What To Explore 857
8.89.3 Discussion / Challenges 858
8.90 Mission name 860
8.90.1 Demonstration Goals 860
8.90.2 What To Explore 860
8.90.3 Discussion 860
9 DML FAQ 861
9.1 General 861
9.1.1 Loading Modules 861
9.1.2 Module Misbehaving? 861

9.2 Module Specific 861
9.2.1 CloneZone 861
9.2.2 Messenger 862
9.2.3 Object Destruct Detector 862
9.2.4 PlayerScore 862
9.2.5 SSBClient 862
9.2.6 stopGap/noGap 863

cf/x

DML

(Dynamic Mission Library)

for DCS

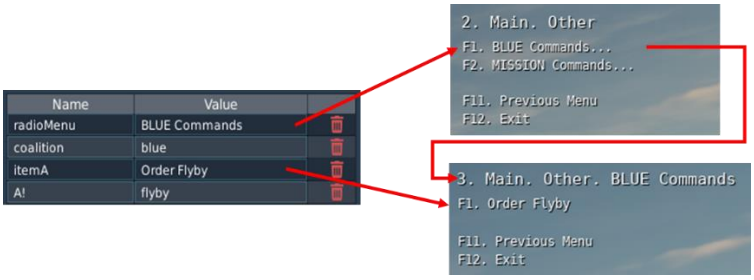
Version II

© 2022 - 2024 by Christian Franz and cf/x

1 Foreword to Version 2

From its beginnings as a small library of often-used helpers to its first steps as a mission designer's collection of ready-to-use functions, to its current incarnation as a "LEGO®"-like collection of "bricks" that designers click together to build great missions, DML has evolved, changed, and grown.

What sets DML apart from many other (often more powerful) mission libraries is its ease of use: mission designers use Mission editor's *trigger zones* for everything, and they never have to deal with a single line of script code (or 'Lua', as mission veterans know to call the Beast by its name). This allows us to use Mission Editor's GUI to place and set up everything – including the way that we tell DML how the various modules ("bricks") should work. Mission designers control even complex aspects with trigger zones inside Mission Editor:



Above example shows how we use Mission Editor to edit a trigger zone to add some "attributes" (*radioMenu*, *coalition*, *itemA*, *A!*) and their respective values. And how they later appear in-game as menus in a players' "communications" menu tree.

This approach of using Mission editor's built-in tools has proven tremendously successful and easy to use. This was by no means assured, and prior incarnations of DML were a bit inconsistent in this regard. Based on feedback from you and fellow mission designers, DML II now aims to harmonize these features and become even easier to use and more helpful. As DML continues to evolve, we'll see more features to emerge, and (hopefully) even better integration that makes creating DCS missions less arduous, faster, and – above all – more fun. Because fun missions should also be fun to create.

1.1 How to get started quickly

DML is big. *This document* is frighteningly big. So how can you get the most out of DML in the shortest amount of time? Like I mentioned, DML is similar to a box of LEGO® bricks: you don't need to know everything there is to know before you can start building and you never use all the bricks at the same time. So it's fine to start knowing just the basics and then learn more as your demands expand.

1.2 What you need to know already

First, I assume that you know how to create simple missions with Mission Editor. You know: create a mission, place a player/client plane, add some units, place a trigger zone. You *must* already know how to do that. Also, you must either watch the 'adding DML to your mission' video tutorial, or alternatively already know how to add a "Trigger Rule" to a mission.

And for running the demos that come with DML you must know how to fly at least one plane. Demo missions in DML use the (free) Su-25T that comes with DCS, so maybe get into that cockpit now, and spend a few minutes with it to set up the controls and get a feeling for it.

1.3 No Scripting, no Lua

Scripting, or knowing Lua, is *not* required. More to the point, **DML no longer actively supports people who want to integrate DML with their own scripts**. You now simply add modules and connect inputs to outputs.

Historically, DML grew out of my own mission design efforts; to this day it provides me the toolbox that I need to accomplish things that are impossible to do with Mission Editor alone.

After DML's initial public release, it became increasingly clear that people love DML's simplicity and "add modules & connect" metaphor; mission designers enjoy to "click together" various modules without ever having to worry about a single line of code. Although DML came with comprehensive scripting support, very few (except me) used it. People love DML's simplicity, and that's where I'm focusing on now.

Therefore, as of version 2 of DML, I have slimmed down the documentation, removed many unnecessary bits involving "architecture" and similar arcane stuff, and focused on the mission design perspective.

1.4 DEMOS!

We are all looking for different things. DML comes with many modules that each add some cool ability to your mission. It's a big, varied toolbox with all sorts of exciting stuff. And like in real life, you'd often only use a few tools per mission. So, what is the best path to using this bog box of treats? You can old-school it by reading this entire manual (and you'd get a nod of appreciation for your thoroughness from me).

Or, you can skip ahead to this manual's Tutorials / Demo section, and then check out the corresponding "*.miz*" mission files in the 'tutorials & demo missions' folder that come with DML. Most demo missions focus on a particular feature or ability. So, look for the demo that most closely sounds like it is what you are looking for. Read the mentioned module's chapters, open the *.miz* in Mission Editor and run it.

Initially, you'll read about some strange things, like these 'attributes' that we add to trigger zones. Just roll with it and see if the demo does what you want. When you want to use it yourself, then revert to the first chapters of this manual that explains the fundamentals of DML (Trigger Zones, Attributes, and Config Zones), and then read the description of those modules that the demo used to achieve the desired ability.

The choice is yours: jump to Chapter 8 *now* - or read on…

2 Preface

Welcome to this document. Thank you for taking the time to actually RTFM – you are wise indeed to read this, as much of what I've written here could make using DML more enjoyable for you and help shorten the time it takes you to use DML in your own missions.

2.1 What is DML?

So, what is DML? It's a **mission-building toolbox**. It contains many "modules" that all can do something for you and that work well together. It's built like a box of LEGO® bricks:

- There are many small *simple building blocks* called 'modules' (e.g.: smoke zone, NDB, Cloner, radioMenu)
- Each block adds some kind of ability to your mission.
- Blocks can connect to each other to build bigger blocks.
- There are also ready-made "big ticket" items that provide distinct feature abilities like CSAR Missions, Sub Hunting, Integrated Player Score and more.
- Most importantly: you seldom need every block for every mission: you can start having fun building stuff knowing just the basics and expand from there.

Here are some "Building Block" examples:

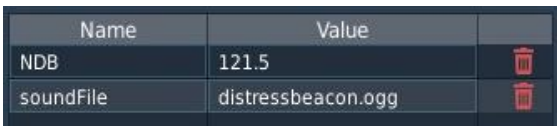
- Do you want to add a moving NDB to a ship? Add the *NDB* module.
- Want to keep kill scores? Add the *PlayerScore* module.
- How about dynamically spawning troops when and where you need them? Look at *Cloners*.
- Your mission needs random civilian traffic? Check out *Civ Air*.

More importantly, DML truly shines when you start putting blocks *together*:

- Do you need targets to spawn on demand? Use a "cloner" for spawning, and a "radioMenu" module to allow players to control the cloner.
- Do you want more responsive airfields? Use a "valet" to greet players when they arrive – and start animating erstwhile static ground forces with the "impostor" module
- Need to choreograph action? Use a "pulser" to provide timed signals, and the "sequencer" to make sure everything happens in the correct order. Then throw in the "delicates" and "ground explosions" modules to blow stuff up on cue.
- Want your own mission to surprise you? Connect a "pulser" to an "RND" module that randomly selects a "cloner" – and with a few clicks you have a mission that even you can't predict.

No scripting. When you add DML, you always add ready-to-use modules to your mission, which you *never edit yourself*.

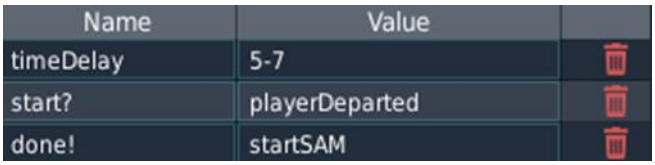
So how does DML work? DML's modules intelligently **attach themselves to Mission Editor's (ME) Trigger Zones**. Mission designers control the module's abilities in ME by adding 'Attributes' to these Trigger Zones.



For example, when you add the above attributes to a trigger zone, the “NDB” 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. If you look closely, you may already make the connections between the various attributes and their values.

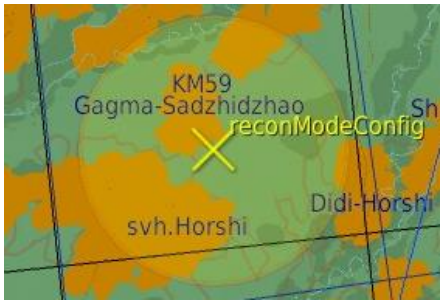
Through this simple mechanism, adding complex new abilities to missions becomes a snap (or, at least, much easier). Since **you control all aspects of DML from inside ME**, you do not mess around with scripts. All DML modules take their settings from Trigger Zone Attributes (see below). You edit those directly in ME: Trigger Zones already have attributes, and editing them is built straight into Mission Editor.

DML uses trigger zone attributes to put DCS mission creation into super-cruise. We can attach new abilities to trigger zones, and control their behavior simply by adding and changing a few attributes. So attributes are used to tell a module how to behave when it attaches to a trigger zone.



If that isn’t enough, DML also uses attributes to allow modules to talk to each other and the mission itself. DML modules have inputs (attributes names that end in ‘?’) and outputs (attributes names that end in “!”) and much of your mission design work is connecting inputs with outputs. DML handles the rest.

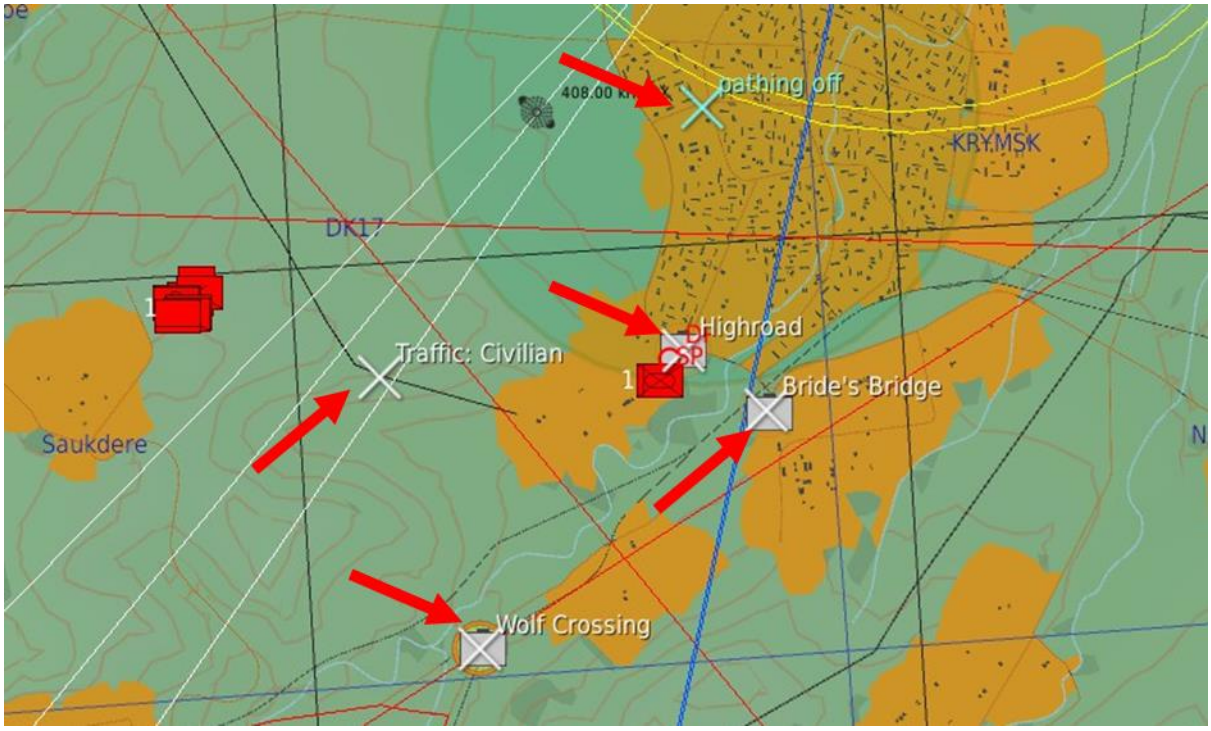
For example, spawn zones can be instructed to watch the flag “playerDeparted” (which is connected to their input), and then spawn enemies every time when that flag’s value changes. Other modules can be told to change the value of a flag connected to the output (e.g., “startSAM”) when they deem it appropriate. For example, the ‘valet’ module will change the flags connected to its output when a player plane enters its zone. This allows you to easily integrate the modules in your normal ME mission design workflow.



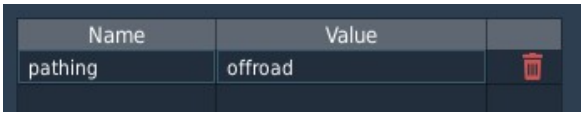
This method of passing data to a module through zone attributes extends to module configuration. In classic frameworks, you would need to change script to courtail it’s behavior to your mission requirements. In DML all configuration data is retrieved from attributes that are added to specially named ‘config zones’. The module starts up with default values, and then looks for its “Config Trigger Zone” that can contain the attributes that you want to change for this mission. **You configure all modules from within ME** – in DML you never change a single line of code.

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 place new functionality simply by adding Trigger Zones to the map and editing attributes. DML's modules automatically home in on the attributes and know what to do.



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 factory zones ("Wolf Crossing", "Bride's Bridge", "Highroad") – these are zones that, when captured by blue or red, automatically produce ground forces to 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 attributes (like "pathing:offroad") to tell DML what to do.

2.2 DML in a Nutshell

So, what's in DML? A big, big toolbox for adding great abilities to your mission. here's a bird's eye view of what you get with DML:

- **"Big Ticket" Modules** ("fully fledged abilities") 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 for aircraft, complete with reporting and custom flags

    - Griefer protection (i.e., protection from asocial people who intentionally abuse your server)
    - A Bomb Range trainer
    - Intelligent slot blocking for preventing spawns on enemy airfields
    - Smart protection from missiles
    - Filling empty player slots with static stand-ins
    - Helicopter Troop Pick-up, Transport and Deployment
    - Score Keeping
    - ASW (Anti-Submarine Operations)

- **Building Blocks** ("modules") that **attach (minor) functionality to Zones in ME**. They provide diverse functionality such as
    - A dynamic messaging system (to "print" dynamic messages to players on the screen)
    - Interactive player menus (communications)
    - Randomizers, Pulsers and other utilities to supercharge your mission flow
    - Counters for events
    - Group Tracking and Unit Detection
    - Dynamic Spawning and Cloning of groups
    - Dynamic Object/Cargo Spawning
    - Artillery Target Zones
    - Factory zones that produce for the owning faction
    - Conquerable Zones and FARPS
    - (moving) NDB
    - Map/Scenery object destruction detector

- **Persistence** – players can **save a mission and then later continue it** (within the confines of DCS's limitations)

- Support for **zone-local flags** so you can stop having to invent new flag names for everything. Keep your flag pool clean when you are using flags to signal within the same zone..

- A **full-bore Interactive Debugger** to inspect flag values, track events, spawn units, report when flags change their value or if their value changes to a specific number – with ME integration that allows you to set up debugging missions before you run them

  [Note: DML's debugger's feature extend far beyond what is listed above, and according to some users is reason enough to get into DML]

- **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 this.

Since DML's first public release in January 2022, a lot has changed: based on community feedback DML has expanded, and added features that I never thought possible: persistence, Stop Gaps, Cloners and The Debugger.

And I hope that with Version II this trend continues – that DML constantly evolves, and there will be more and new avenues to explore. Based on feedback from you and others, I expect DML to continue to grow in new and exciting ways.

I hope that you'll enjoy the ride as much as I do!

Zürich, January 2024

-ch

2.3 About this Document

This document is divided into multiple parts:

- **Part I: Introduction & Overview**
  This 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 'a DML primer', it shows you – in a friendly and hopefully understandable ways – how you add modules to your mission, and how you use those modules in your missions. We run through some effects and then control one module with another one to create fire effects.
  We briefly touch upon the main parts of DML, how the pieces fit together – and we even showcase a couple of the most important modules – the "crown jewels":

- **Part III: Reference**
  This is the look-up part, less friendly but very accurate: every single module has its own part, and I describe in detail how you could use it in your mission. Each and every attribute that a module supports is meticulously listed and explained here. Unless I forgot.

  When you read through this part of the manual, pick only those modules that catch your fancy; all module descriptions are structured identically and should be self-contained; they can be read in any order.

  This section is split into "Basic Building Block" Modules and "Big Ticket" full function items.

- **Part IV Persistence**
  DML sports an ability that's often sought after: the ability to write missions to storage and continue later. DML can do that for you, and this chapter tells you in detail what that requires from you (hint: not a lot).

- **Part V: The Debugger**
  Deep beneath the surface of DML lurks a highly evolved predator that feeds on bugs. Get to know your frighteningly powerful, yet always willing-to-help apex predator.

- **Part VI: Tutorials & Demos**
  DML comes with a comprehensive, sometimes even 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.

  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 designers, 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 will jump-start your own mission designs with DML: many demos focus on a module's features, and often show how modules work together to build a better effect. When you are done putting the mission through its paces, read the 'Discussion' part for each demo to find some interesting point you may have overlooked.

Neither part is meant to stand on its own; parts I and II are too short to convey much helpful information, but they will get you started, and make you familiar with some concepts that may otherwise sound strange. Part III, on the other hand, provides an ocean of detail that you can drown in. So get the big picture in parts I and II, then skim part III. Should you get lost in the details, perhaps refer back to part I and II, get your bearings, and then head out back on track. Or skip ahead to **part VI** – the tutorials and demo missions often explain much better how things fit together than the arid stretches of the wastelands called Part III.

And – explore! There are demo missions to investigate and play around 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.

2.4 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.

2.5 Copyright

This document, and all accompanying code and demos are copyright © 2021 - 2024 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 © 2022-2024 by cf/x and Christian Franz"

in that work's documentation.

For commercial use, please contact me.

2.6 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)

cf/x

DML
for DCS

PART I:

INTRODUCTION

3 Introduction to DML

DML is a collection of tools that greatly enhance the scope for missions that creators can build in Eagle Dynamic's "DCS World" Mission Editor. It instantly adds direct access to new abilities like unit spawners, artillery bombardment, bomb ranges, civilian air traffic or CSAR missions into your missions. Mission Creators add these features to their mission selectively, based on their demand, and can later easily change this.

DML comprises of:

- **Basic "Building Blocks" (Modules)**
  Modules in DML all provide some functional building block that can stand on its own, and often is used in combination with other modules to weave even greater abilities. For example, the *'Valet'* module can detect when a player arrives inside a zone, and then tell the *'Messenger'* module to greet the player.

  All DML modules are attached to trigger zones with Mission Editor, and use Mission Editor "Attributes" to control how these modules work for the zone that they are attached to.

  Another way to look at this is: **Zones** tell DML *where* something should happen (inside the zone), and their **attributes** tell DML *what* should happen and *when*. For example, attributes tell a "Spawn Zone" *which* units to spawn and *when*. Should the units spawn, they spawn inside the zone (*where*). Attributes describe to the "Smoke Zones" module what color the smoke should be in, and they control how the "Civ Air" module allows air traffic to flow.

  Using Trigger Zones with attributes has numerous enormous advantages over classic purely script-based solutions:
  - (obviously) 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 never concern designers) from visual mission building.
  - we have a graphical representation (the enhanced zones) of where we place the new functionality. You can use click & drag to reposition and resize the area where a module works.
  - we can use copy/paste to quickly replicate zones with their attributes over the map.

  All this seamlessly integrates DML into your familiar mission editing workflow.

- **"Big Ticket" Full-Feature Modules**
  These are simply 'bigger blocks' that provide full-feature abilities in a drop-in package. They add complete, new abilities (e.g. CSAR -- Combat Search And Rescue) to your mission.

- **Zone-based configuration and data access**
  Modules also use Mission Editor's Trigger Zones to open their configuration details to mission designers. That way you can change how modules behave in general without having to access or change the underlying code.

- **An interactive Mission Debugger**
  that is fully capable of blowing your socks off.

- **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; DML is self-contained. No framework (e.g., 'MIST' or 'MOOSE') is required; there are no known conflicts when you run other frameworks side-by-side with DML.

3.1 DML Modules

The entry barrier to use DML is low: it requires that you are able to perform the following (quite fundamental) steps from within DCS Mission Editor (ME):

- Place Trigger Zones, and
- Add/edit/remove "attributes" to/from Trigger Zones.

Beyond that there are no requirements; **DML does not require Scripting nor Lua knowledge**.

For a typical mission building session, once you have decided what kind of mission you want to build, you simply determine which modules you need to provide desired abilities (e.g., you might want the ability to dynamically spawn units, provide a player menu, randomize some events, and conduct CSAR Missions), add those modules to your mission, and then start laying down trigger zones with attributes.



In above example, we see two small trigger zones "Countdown" and "Zero Message" that I placed in Mission Editor. I then edited those zones and added some attributes to each (can't be seen in the image above): The "Countdown" zone received attributes to control a "countdown" module, and the "Zero Message" zone received attributes to control "messenger" module

DML's modules are grouped by their scope size:

- “Building Blocks” are general purpose modules that provide some generic ability that is easy to understand, for example ‘Smoke Zones’, ‘NDB’ or ‘Clone Zones’. They are designed to communicate with each other, and allow mission designers to link them together to build bigger abilities.

- “Full Feature” (a.k.a. “Big Ticket”) modules are quite similar, except that they deliver entire abilities (like CSAR Missions, Player Scoring, ASW) as fully customizable drop-in modules. They interface with ME exactly like “Building Blocks”: through trigger zones and attributes – so if you know how to use a building block, you already know how to use a big ticket item.

- Additionally, DML provides some modules that can enhance the multiplayer experience and, in order to function, must be added the DCS’s server.

So let’s take a tour through DML, and just briefly stop at each attraction. Part II will go into detail, so for now let’s get the Big Picture: how does DML work with ME, and what does it offer to mission designers?

3.1.1 “Building Blocks”

The bulk of DML consists of small “modules” that each add specific abilities to a mission, like placing colored smoke, spawn units, or set up beacons for ADF or TACAN.

Some of these modules concern themselves with changing “flag” values. Initially, if you don’t know why that may be helpful, simply skip those. Eventually, when your mission designs become more complex, you’ll discover flags, and suddenly, these modules’ benefits will make sense to you. You won’t need them before that.

To use modules, mission designers place a Trigger Zone, and then use ME to add the required “Attributes” to these zones.

So it’s time for a little “show and tell”: here’s an overview of the basic “Building Blocks” in DML:

Module	**Features**
airfield	Controls **ownership of airfields**, provides ownership info to other modules
artilleryZones	This simulates artillery target zones for interaction with FO. Can **simulate artillery bombing**.
artilleryUI	Provides a **player interface for Artillery Zones**. Allows helicopters to request smoke marks on the target, and when close enough and in direct line of sight (LOS) to the zone’s center, pilots can order the artillery to fire into the zone.
baseCaptured	A module that generates **output** signals (so other modules can react) **when a base** (airfield, FARP) **changes hands**.
cargoReceiver	These zones work in conjunction with the CargoManager module. **Delivering** (unhooking) a helicopter’s **sling-loaded cargo** in such a zone **triggers their output**.
changer	A module that provides **on-the-fly flag value conversion** and can function as a **gated switch** for module inputs.
cloneZones	One of the must-have, and most capable DML modules. Like Spawn

Module	Features
	Zones this module adds the ability to **dynamically spawn units**. Clone Zones create groups **from templates** that are easy to build in ME. Spawning occurs on demand or automatic.
convoy	Allows mission designers to easily create (**helicopter-escorted**) **convoys** that can be started at will and integrate with DML by sending signals for all important events. Has optional reporting abilities.
countDown	A very flexible **counter module that counts how many times its input was triggered**
counter	A simple module that does exactly what it says: **counts** in one way or **another the times it has been triggered**.
delayFlags	Smaller sibling to pulseFlags, use this module to introduce a **delay on the output.**
delicates	Makes units and objects inside 'brittle' so they **explode when they are hit**
factoryZones	Factories are zones that **produce units for the faction (red/blue) that owns the zone**. As such, factories are an extension of Owned Zones. They usually provide a central imperative for missions to conquer them.
FARPZones	Adds the ability to **better manage FARPs capture** and provide ownership information to other modules. It also provides production of defenders and It ensures that all resources for reload and repair operations are available to players. Ownership of the FARP itself is managed DCS.
fireFX	Big brother to the smoke zone. Flames optional. Handles **multiple flames, randomized** locations
flareZone	Allows you to **launch flares** the DML way: randomized, multiple flares, the full Monty.
groundExplosion	Make **things go boom inside** the zone whenever you want, with more variety than you can shake a dynamite stick at
groupTracker	Have DML **watch your groups** and set output flags when something changes
impostors	A page from advanced game programming gurus: it gives (performance high-cost) AI-controlled **units** in this zone the ability **to turn into (low-cost) static objects and back** at will.
LZ	Whenever you need **a landing or take-off inside a zone to trigger** something, call LZ!
messenger	One of the central DML modules - a DML-style version of ME's good old MESSAGE TO XXX and SOUND TO XXX actions – with inputs to trigger and advanced wildcard processing: **send dynamic text** to all, a coalition, a group, or unit.
NDB	**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 the input**
objecctDestruct detector	This little gem's goal is to provide you with a simple method to know when a **Map Object is destroyed** – be it a bridge or building
objectSpawn Zones	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 use this enhancement to dynamically spawn cargo for helicopters to sling-load.
ownAll	Provides complex, even hierarchical control structures for **managing ownership information** (from owned zones, FARPS, airfields) and sendins singnals on outputs when conditions are met
ownedZones	This adds the ability to create **zones** (areas on the Map) **that can be captured**. Capturable zones are a central tenet of many complex multi-

Module	Features
	player missions and a fun way to create goals for any mission. OwnedZones make the ownership information (who owns the zone) available to other modules via outputs
persistence	This module adds the ability to 'persist', i.e., **write the mission's current state to file ("save")**, and then later load that state back into your mission. This module handles loading and saving mission data for all other modules: any module that loads after persistence automatically uses persistence when it is appropriate.
playerZone	Allows you to **detect player-driven events** like "n players are currently inside this zone".
pulseFlags	A module that **repeatedly changes ME flags** – multiple times, for as many times as you want, at your own pace and time, with (of course) controllable methods to start and stop
radioMenu	*The* quality-of-life module for mission designers that by itself makes it worth adding DML to your mission. It adds the ability to easily **create interactive in-game player menus** with up to four items per menu.
radioTrigger	Some Quality Of Life module to better **use Mission Editor's "Communication"** radio bound flags with DML modules
raiseFlags	A module to **set flags to a value** - at mission start or a (randomizable) time after that.
rndFlags	A simple way (finally!) to **randomly set 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.
sequencer	*Everything* is a sequence: things should happen one after the other. Use a sequencer to take the tedium out of **ensuring that** yes, indeed, **A does happen before B**, dammit!
simple mission restart	An easy way to **restart your current mission** on command, even when on a server, no matter what it is named.
smokeZone	Adds a **permanent, colored smoke effects** to the center of the zone. It doesn't stop smoking unless you tell it to. You control smoke color with the 'smoke' attribute's value.
spawnZones	This adds the ability to **dynamically spawn troops** in a zone - automatically, and on demand. Spawns can occur multiple times. Spawned troops can be issue dcomplex orders
sweeper	Remove non-moving units from inside a zone to **avoid traffic jams**, especially on airfioelds
tacanZone	Place **TACAN nav aids** where you please, and even randomize them.
twn	Looks up **map locations** and replaces their coordinates with nice, human-readable text like "1 nw SW of Zugdidi"
unitPersistence	While many DML modules automatically handle **persistence** automatically for whatever they are handling, there's a glaring hole**: units and static objects that you place with ME**. UnitPersistence takes care of that for you.
unitZone	**Trigger Zones done right**: set all the attributes right there on the zone and drive outputs (flags) the way god wanted you to.
Valet	**Greets and good-byes player units** when they enter or exit a zone. Lot's a nice features, advanced text messaging
WHpersistence	Transparently adds **persistence for warehouse** inventories.
wiper	A DML module to on command **remove objects, units, and other stuff** inside the zone
xFlags	A module to **combine multiple flag inputs for decision making** ('one of these input flags is true?) or to use as a **'gated switch'** (i.e., an output that can be blocked/turned off)

3.1.2 "Big Ticket" Modules

While basic "Building Block" modules are designed to aid mission designers in their quest to assemble a bigger whole, these "Big Ticket" modules add fully fledged 'drop-in' functionality to DCS missions. Their abilities, too, are placed with trigger zones, and are highly customizable – again through attributes. Some of these big-ticket items will automatically interact with, or expand, the capabilities of other modules (for example, CSAR Manager, Helo Troops, autoCSAR and Player Score can all augment each other and automatically integrate with spawn zones and clone zones).

This is what's available:

Module	**Features**
airTank	Adds helicopter/aircraft **fire-fighting abilities** to your mission. Works in conjunction with the **inferno** module.
ASW	Provides **Anti-Submarine Warfare** ability to players: collect and drop sonobuoys and torpedoes to hunt submarines. Includes live updates on the F10 map.
autoCSAR	**Automates the creation of CSAR missions** for csarManager: whenever a pilot ejects (AI or Player), autoSCAr generates a CSAR mission for the coalition that the ejecting pilot belongs to.
bombRange	Fully fledged **training bombing range**, complete with statistics.
civAir	This module provides **AI-controlled civilian (well, neutral) air traffic** that flies between airfields in the region. Provides optional support for the CAM expansion.
civHelo	Provides **civilian** (non-aggressive) **AI helicopter traffic** to your map. Flights depart one civHelo zone and fly to another civHelo zone
csarManager	A drop-in feature that **provides CSAR Mission** support: pick up downed pilots and deliver them to safe zones. Provides a convenient and easy ME interface to create CSAR missions in ME.
guardianAngel	A module that **destroys missiles inbound** on certain airframes just before they hit. Will give statistics about missiles dodged. Can be used to simulate 'jamming' of missiles.
heloTroops	A drop-in feature to enable **player-controlled troop** helicopters to pick up and deploy infantry. Can interact with spawn zones and cloners to request troop production.
inferno	Create **large, spreading fires** on the map that **can be extinguished** by players using the *airtank* module
limitedAirframes	This module provides two significant additions: - **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).
noGap	**Alternative to Stop Gap**, multi-unit player group capable. Not as reliable as Stop Gap due to design issues with SSB
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.
PlayerScore /	Provides **score-keeping, feats** and **kill-tabulating**, fully MP-capable,

Module	Features
PlayerScoreUI	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 module PlayerScoreUI
reaper	Allows mission designers to easily add **reaper drones** that report, track and **lase targets**
reconMode	A module that allows planes (AI and Player) to **automatically report enemy groups and mark them on the F10 map** for all same-faction players. Supports priority- and black-listed groups.
scribe	This module **records all players accomplishments** (engine start-ups, time in a type, landings, take-offs, crashes) and can – per player – give them a summary. Supports persistence so a player can see their accumulated stats **across multiple replays**.
shallows	**Removes** destroyed **ship hulls** after they fail to sink (usually happens in harbors).
sittingDucks	Extends StopGap so that when **a player stand-in aircraft is destroyed, the player slot is blocked**.
slotty	A module that disallows players to slot into aircraft that are currently off-limits to them (for example if they are blocked by ssbClient or ssbSingleUse). Works on single- and multiplayer missions. Does not require SSB be installed for multiplayer missions, and works nicely alongside it.
ssbClient	A module that allows **slot-blocking** for **aircraft** 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 or the mission is using ‘slotty’.
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)
stopGap	**Fills player slots on airfields with static stand-ins** until a player jumps into the plane.
taxiPolice	A module that **enforces a speed limit on taxi ways**. Repeat offenders are punished by rendering their current aircraft unflyable (adding 200t of weight)
TDZ	Fully fledged **training tool for landing** aircraft
unGrief	**Griefer-repellent** for your server: makes it undesirable for **players to kill their own** side.
williePete	This module allows players to **mark artillery targets** with WP (“Willie Pete” in old phonetic alphabet) rounds and then **call in artillery** strikes on the marked target.

3.1.3 Server Modules

DML provides several modules that can be installed to enhance server operation of multiplayer missions. They provide additional abilities to missions, greater stability, or entirely new functionality.

Module	Features
noGapGUI	Provides enhanced compatibility and works around a DCS server bug for missions that use the 'noGap' module
smrGUI	Provides the server-backend functionality for those missions that use the simple mission restart module
stopGapGUI	Provides enhanced compatibility and works around a DCS server bug for missions that use the 'stopGap' module
Tickle	Provides anhanced, automatic mission rotation on a server and an interactive interface for authorized players

3.2 Other Attractions (Overview)

The modules (both basic building blocks and full feature drop-ins) are what you would use in your missions to create better content. To round out its features, DML also comes with

3.2.1 The Debugger

This is a big-caliber development tool not designed for the faint of heart. Filled to the brim with Quality of Life features and abilities to make 'plain vanilla' mission designer green with envy, it packs enough power to become dangerous when wielded carelessly.

Here's a quick rundown of its benign features:

- Monitor mission events
- Watch over flags and report when they have changed
- Create flag picture 'snapshots' and compare them
- Inspect flags

Some cool features include

- Spawn a variety of units directly on the map
- Remove groups or units
- Integration with ME to allow debugger setup in ME
- Write the entire debugger transcript to a text file
- Inspect Lua variables

And this is where it becomes potentially dangerous

- Create, set and delete Lua *variables in the currently running Mission Scripting context*. Yeah. "Do you feel lucky, punk?"

3.2.2 Persistence

DLS comes with built-in full support of persistence – the ability to 'save' a mission to storage and then 'continue' it later. The quotes have a reason: DCS itself imposes very strict limits to

this ability. Inside those, however, DML can provide your mission with some great ways to save & continue.

3.2.3 Multi-Player support built in

All DML modules fully support multiplayer, there is no need for you to build separate single-player and multiplayer versions.

3.2.4 Fully Documented Demos

There is at least one demo for each module, and for each demo there is one dedicated chapter in this document. In fact, the demo documentation is bigger than the module documentation, so make sure that you at least take a look at what is offered.

PART II:

USING DML

4 Using DML in your missions

DML enhances normal mission design with 'vanilla' Mission Editor in that you lay down some trigger zones and then add some attributes to those trigger zones. There's little else involved. So, let's look at DML from a mission builder's perspective, and how you would typically use DML in your mission building efforts

- *Design the Mission in your head*
  Like any mission, you first come up with an idea, and you decide upon the mission's main features. Unlike with plain ME, with DML you make some initial mental notes which modules deliver what functionality / ability (e.g., CSAR, radio menus, conquerable zones, Player Score) that you want and incorporate them into your mission design. Since DML is modular, you usually need a handful of modules per mission

- *Create a new mission / open existing*
  Create a new mission (or open an existing that you want to add some DML abilities to – like CSAR missions), and start editing/building.

- *At any point: add DML Modules to your Mission*
  Doing so takes roughly half a minute, and you usually bring only those modules into your mission that you need at that point. It doesn't matter when you do it (and you can always add more modules later). I usually do it directly after choosing a map.

- *Complete your mission, add more DML features where needed*
  Build your mission as always. To put down some DML functionality, you usually simply add a trigger zone, and add a couple of attributes in ME's zone editor.

- *Test, rinse, repeat*
  Run the mission, and test if the DML modules deliver what you want, perhaps tweak some attributes. When debugging a particularly difficult issue, remember that DML comes with a high-caliber debugger that can help you track down stubborn issues.

4.1 The Basics

4.1.1 Installing DML

DML has no preferred spot that it must be installed in, and you can download and put it wherever you feel is best. During mission creation, you will occasionally go to wherever you stored DML and copy some text and paste it into your mission. That's all. There is no installation procedure.

4.1.2 Add DML to your Mission

Like a LEGO® set, DML comes as a chaotic, colorful heap of "*modules*". They come in many, sometimes odd shapes. With some modules, we see immediately what they do, while others appear mysterious - and you can put those to the side.

Strange or not, when it comes to adding them to your missions there's a simple fact: all modules are nothing more than just text files. These text files are named for what they do, with the extension 'lua' instead of 'txt' to show that they are written in a programming

language called ‘Lua’. They can be opened with any text editor. So, the “fireFX” module would create fire and smoke effects inside a zone, and it resides in a simple text file called “fireFX.lua”. And here’s the good news: once you have added a module to your mission, you won’t ever deal with it directly again.

You build your mission and add only those modules to your mission that you need, when you need them. Usually, you only need a few. Since *“dcsCommon”* is the first module that you bring into your mission (along with *“cfxZones”*), we’ll show how to add these two. Adding the rest works exactly the same way.

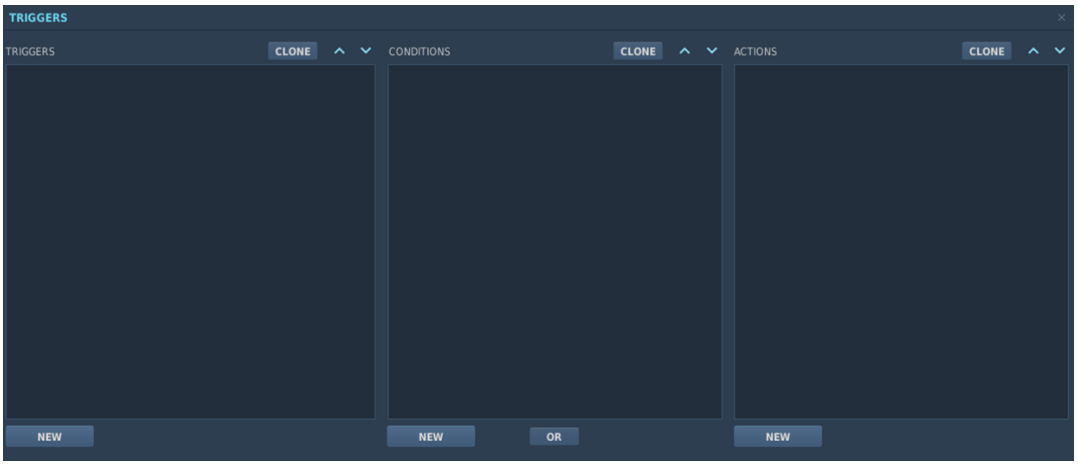
To bring the modules *“dcsCommon”* and *“cfxZones”* into your mission, do the following:

In Mission Editor,

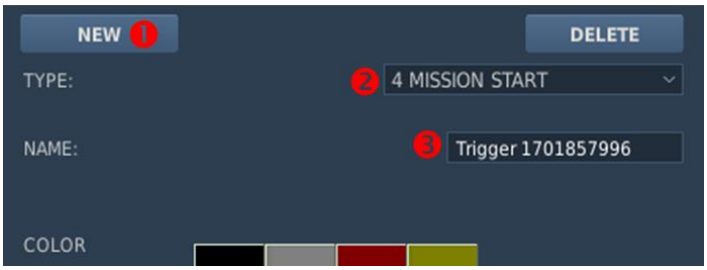
- Click on the “Set Rules for Trigger” tool



- A (really) big (and intimidating, don’t be afraid) panel opens. It’s divided into three panes “TRIGGERS” (left), “CONDITIONS” (middle), and “ACTIONS” (right). The UX for this panel is decidedly last-millennium 90’s, and it may even appear hostile compared to modern interfaces.
  The way that it works is very “old-school”, and mostly works ‘from left to right’: whatever you do in the leftmost “TRIGGER” pane affects all the panes to the right (CONDITION and ACTIONS). It just does, there’s no requirement to understand why, just accept it as a way to do things in DCS.

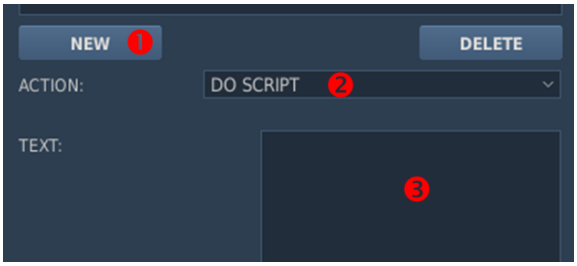


- Now, below the “TRIGGERS” pane, locate and click on the button “NEW” (and yes, there are two identically labeled buttons “NEW” below the two other panes to the left. Let’s simply ignore these crass UX blunders, they merely add spice to working with ME). This creates a new “Mission Trigger and Rules” entry which we will use to load DML. See ❶ in below image



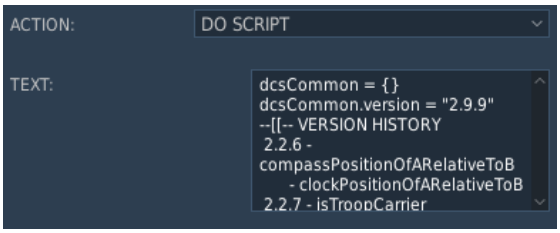
After you click NEW, more fields appear below the NEW button. From the TYPE drop-down (which should currently read “1 ONCE”) select “4 MISSION START”. This is ❷ in above image, and it tells DCS that this trigger is to be executed when the mission starts up – we want DML to load at mission start.
Optionally: you can ❸ change the name of this trigger. The name initially assigned is randomized, so what you see can be very different from above image (where it happens to be “Trigger 1701857996”). Use any name that you like, I usually choose “Load DML”.

- We created a trigger that always runs when the mission starts up. We now tell DCS what to do at mission start. We do this via the ACTIONS pane on the right. Make sure that the left pane still shows the “4 MISSION START” info that we just added, and then ❶ click the NEW button on the ACTIONS pane.



- When you click on “NEW”, new items appear. One of these new items is the ACTION pop-up. Click it, and ❷ choose “DO SCRIPT”. This tells DML that when this “trigger is run”, the first action should be to run a script. Which script? Read on.

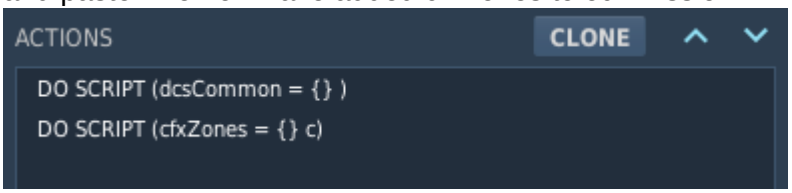
- Locate the small box named TEXT. The Text inside this little box is the script that DCS runs when the mission starts. Remember when before I told you that all modules are just text files? Well, now go to Windows Explorer, open the “modules” folder from wherever you put DML. and look for the “dcsCommon.lua” module (text file). Open it in your favorite text editor (e.g. Notepad), and select all text (use the shortcut CTRL-A), then copy the text to the clipboard (shortcut CTRL-C).
Switch back to DCS, click into the text box ❸, and paste the previously copied text into that text box (CTRL-V). You now have added the “dcsCommon” module to your mission.



- Let’s continue and add another module “cfxZones” to our mission. The trick here is that we don’t have to create another trigger, we simply add another ACTION to the

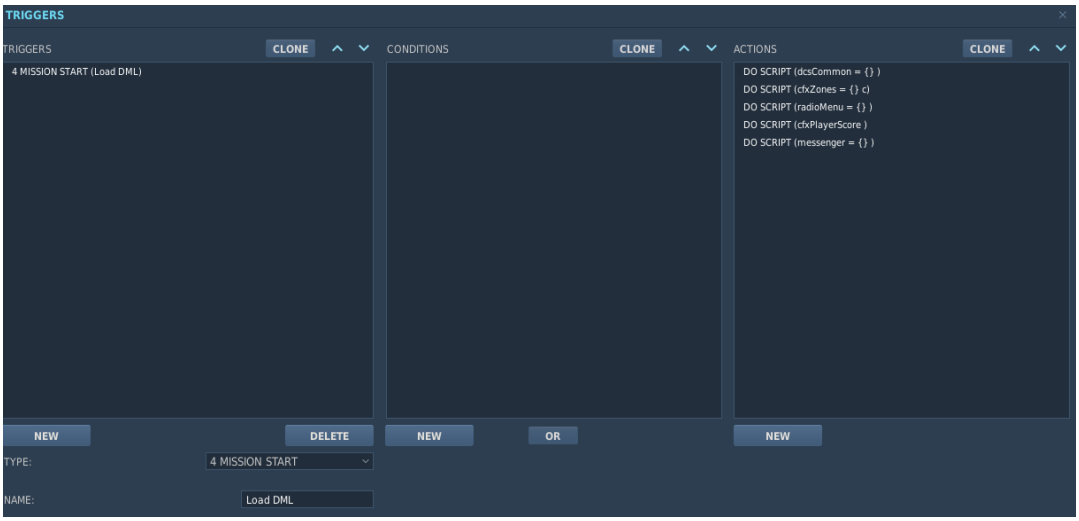
one that already has added dcsCommon: Click on the same NEW button in the left ACTION pane.

- Again, choose “DOSCRIPT” as ACTION, and then go to windows explorer to find and open “cfxZones” in your text editor. Select all, copy, switch to DCS, click into box, and paste. We now have added cfxZones to our mission



- Rinse and repeat for as many modules as you want to add to your mission.

- If, at some later point you want to add more modules, come back to this trigger, and add more ACTIONS to it.



In above example, I have added five modules to my mission: dcsCommon, cfxZones, radioMenu, cfxPlayerScore and messenger.

Note:
We use DOSCRIPT Actions here because I like their simplicity. Other mission designers prefer a DOSCRIPTFILE Action instead. Either will work, and for simplicity I’ll stick with DOSCRIP for the entirety of this document.

Note II
You can add DML at any time to a mission – you do not have to add these modules before you start designing, and you can easily add DML to existing missions. DML plays nicely with other frameworks like “Mist” and “MOOSE”, so you can easily add some DML magic to missions that are built on those.

Note III (advanced users)
You may wonder why we don’t simply pack all DML modules into one big file that simply

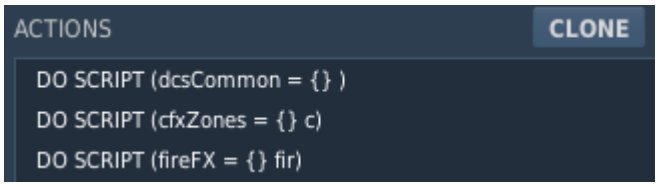
works for all missions with just one DOSCRIPT action. Doing that will surely work, and if you like that simplification, feel free to do so. The performance penalty is negligible. From an engineering perspective, however, it's highly displeasing and inefficient to create a monolithic slab out of what is architecturally designed to be beautifully modular.
A definite disadvantage of this method would be that each and every time that a new version of DML is released – a new function added, changed, or bugs fixed, you'd have to build an entirely new file, and bring that into your mission.

4.1.3 Adding Building Blocks to your mission

Let us assume that you have just added the two DML essentials "dcsCommon" and "cfxZones" to your mission as described above. Those two modules are "DML Bedrock" – they won't give your mission any new abilities, they provide all other modules access to DML's superior abilities to handle zones, attributes, an interface with DCS.

So, let's imagine that we want to use the "fireFX" module: we want to add some flames and smoke to Batumi's airfield. Agreed, you can do something similar with ME only; this is merely a somewhat contrived demo example, plus DML modules are much easier to use and versatile (as we'll see later).

So the next step is to add the "fireFX" module to your "4 MISSION START" trigger. I'll leave that to you as an exercise. The resulting ACTION pane should look like this at the end:



4.1.4 Placing DML Modules on the map

Now that we have added the "fireFX" module to our mission, let's try using it. From the description (see the 'Reference' section, later in this document) we know that fireFX zones place one (or more) fire effects in the trigger zone that we attach the module to. Unlike the "Effect Smoke" action that is available in plain vanilla DCS, DML fireFX provides you with simply means to randomize the number of effects and location inside the trigger zone. I often use a single fireFX zone over an area with multiple randomized fires to create an element of visual random and chaos.

Here, in this demo, we start by learning how to attach a DML module to a trigger zone, and how to use attributes and 'defaults' to control our results. In Mission editor, I place a trigger zone squarely in front of my 'client' player aircraft (I chose a Hog that "Take off from ground hot". Use any plane that you prefer), name it "You're fired", adjust its size to 20 feet, and color it blue for better contrast.

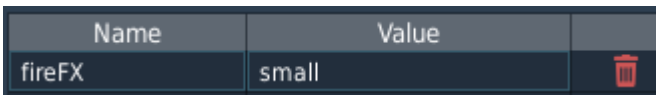


So far, standard stuff. Now let us attach a "fireFX" module to this trigger zone. When you look up the "fireFX" module's reference, the "ME Attributes" section tells us that the attribute "fireFX" tells DML that this is a fireFX zone. It also tells us that we can add a value to this

attribute to set the size of the effect. If we do not supply a size, fireFX defaults to ‘small’ as size. Let’s decide on ‘small’ for size.

To tell DML that we want fireFX to erupt from the this zone, we

- Click on the Zone (“You’re Fired”)
- Click on the “Add” button below this zone’s **Attribute Editor** (don’t panic, see below)
- A new row appears in the table with the word “PROPERTY_1” in the Name column. Replace the word “PROPERTY_1” with the word “fireFX”. This tells DML that you want to attach a DML fireFX module to this zone

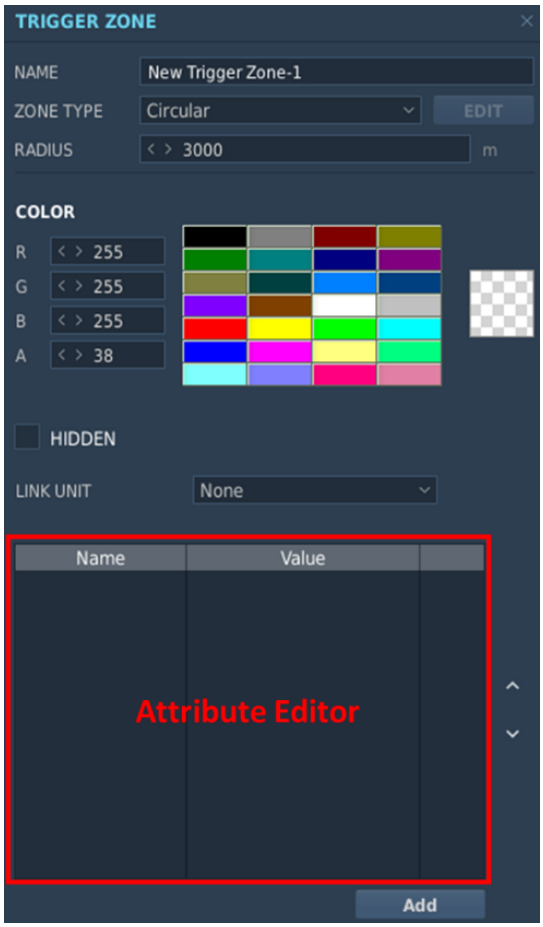


- In the VALUE column to the right, add the word “small” – this tells DML that the effect(s) placed in this should be small. Strictly speaking, we could have omitted ‘small’ since this is the default, but for demo purposes, we’ll be extra careful.

We are done – except for one big question…

4.1.5 Wait… Attribute Editor? What’s that?

If you have ever created a Trigger Zone, you have already seen their zone attributes and their Attribute Editor that come with it. You likely ignored them because – before DML - they have had little practical use.



Go into Mission Editor and create a trigger zone – anywhere on the map, or click on an existing trigger zone. As soon as you place, or click on, a trigger zone, the right side of the map is replaced with the familiar Trigger Zone Editor. Is shows controls to change the zone’s size, shape, name, color and more.

And below all that sits a strange, big box that contains a table consisting of a couple of weird, empty columns headed “Name” and “Value”.

In geek parlance, the combination of a name and value is often called an ‘Attribute’: something that can be named, and that named thing has a value. That’s all. A posh name for something very boring – until you need it, that is. And in DML we’ll be using these attributes a *lot*.

So, inside this red box we can add, modify, and remove “Attributes” to and from any trigger zone in a mission. Even better, this editor was there all the time, so DML fits right in with DCS and Mission Editor. Just before, we added a “Name and Value” pair to the Trigger Zone named “Smoke’m”: an attribute named “smoke” with the value “orange”. When the mission starts and runs DML, it scans all trigger zones for module-specific keywords. Should it find a match, it tells the relevant module about what it found, and the module does it’s thing. “smoke” is a keyword that activates the smokeZone module

for this trigger zone, and the value ‘orange’ tells the module what color to use. The trigger zone itself tells the module where to place the smoke: at its center.

Long story short, let’s enter out aircraft and see if it worked. Save the mission and start it by clicking on the green ‘Fly Mission’ button on the left, then enter your aircraft.

4.1.6 Attributes and their Defaults

If you did everything correctly, you should be in for a big disappointment: you’d see something similar to this (I switched to F2 outside view, zoomed out and rotated the camera a bit): a big nothing.



No fire, no smoke, nothing. Why? What went wrong?

This is easily explained, and *nothing went wrong*: the various modules in DML come with a host of options to make them as versatile as possible. However, to be easy to use, and to spare mission designers some hassle, they all “assume” certain settings unless you say otherwise. This is called “assuming a default”, and each module assumes a default value for any attribute that you do not explicitly add to a trigger zone. The idea is that mission designers should be able to use modules while having to add as few attributes as possible. You only add an attribute if your intent to use it differently from what is the default.

If you look through fireFX’s documentation, you’ll see that is has an attribute “onStart” that defaults to “false” – DML assumes that you, the mission designer, want to start the mission without the fire(s) burning inside the zone, and start them yourself later, on-demand.

	Defaults to <none>
onStart	If set to true, the mission starts with the effect on Defaults to false (no effect on start)

So, in order to start the mission with the fire(s) a-burning in this zone, you have to change the value of *onStart* (which defaults to “*false”* to “*true”*). And speaking of flames, we see that fireFX has an attribute that controls flames, the surprisingly named “*flames”* attribute. *That* one defaults to *“true”*, so we do not have to change it.

flames	If you supply “false” for this attribute, only smoke (no fire) is displayed. Defaults to true (flames and smoke)

So we only need to add the “onStart” attribute to the “You’re Fired” zone; before we do that, here’s another important tip:

4.1.7 That text in the upper right corner…

When you started the mission, some messages appeared in the upper right corner of the screen that look like this:



These are DML's 'startup-line" diagnostics that each DML module writes to the screen when it starts up. Is it important? Yes. It tells you which modules you have successfully added to the mission and each module will also complain if there is anything is amiss. Check this while designing a mission religiously, because this text can tell you if you made some misconfiguration, and it can be the first indicator if something is wrong.

Incidentally, it shows a warning for the way that we are using the fireFX module:

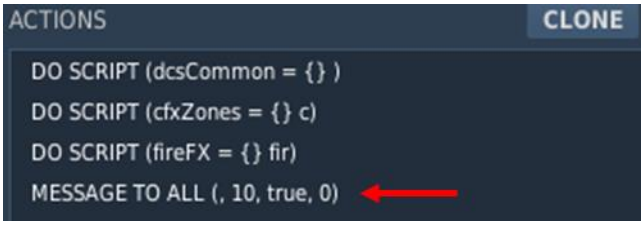


And indeed, fireFX did not respond as we expected. Now, DML does its best to perform what we call 'sanity checks', and out current use of fireFX fails that because we have just placed a fireFX that defaults to 'off', and that has no way of starting. DML can detect that. Just don't rely on DML being able to detect everything that can go wrong. But these startup-lines can be of great help, they often are your first help when a module doesn't respond in your mission. You may laugh now, but when you test your mission and you do not see a module do what it was supposed to do, check the startup-lines to look

- for warnings, and failing that,
- if you have added that module (I've forgotten to add an important module more times than I care to admit), and if it started up correctly (sometimes a module relies on the presence of another module, and if you did not include that module, the 'dependent' module can't work either).

Remember that you can also recall these startup-lines in the "Messages History" window that's accessible at any time in the mission when you press "ESC".

So, if you are done designing your mission, you may want to get rid of those messages because they muck up your carefully crafted scenery. Is that possible? Yes, easily:



When you are done adding all DML modules in the AT START trigger rule (see "Add DML to your mission" above), add a MESSAGE TO ALL Action as a final action, leave the text empty, and put a tick in the CLEARVIEW box. That will erase all messages from the screen – but they remain accessible through the 'Messages History" view.

4.1.8 Adding more attributes

So, coming back from all these detours, we now remember that we need to add an attribute named "onStart" to the "You're Fired" trigger zone. From the fireFX documentation we find that if we want the fire effects in that trigger zone to be active on mission start, we need to set that value to true.

To add an attribute, we click on the “ADD” button below the table, and replace the new “PROPERTY_2” with “onStart” for Name, and “true” for value (note that instead of the word “true”, DML also accepts the word “yes” as a QoL feature” – as shown in the image on the right).

Name	Value	
fireFX	small	
onStart	yes	

Run the mission again:



Finally. And yes, that’s what DCS deems a ‘small’ fire. Remember that fire effects are visual eye candy only and cannot hurt units.

So, attributes help refine what a module does inside a trigger zone. And they have another, incredibly important function that exponentially raises DML’s abilities: they allow trigger zones (or rather, the modules that anchor there), talk to each other. And DML modules love to talk to each other, to tell each other what is happening, and command each other what to do.

There are multiple ways to look at this ‘talk to each other’ ability: the “Yeller Model”, which is a simplification but works 99% of the time, and the much more technical “Flags” model, which is quite a bit more advanced and difficult to understand. I recommend you start (and stick) with the Yeller Model. The rest is still there if your needs outgrow that model’s abilities.

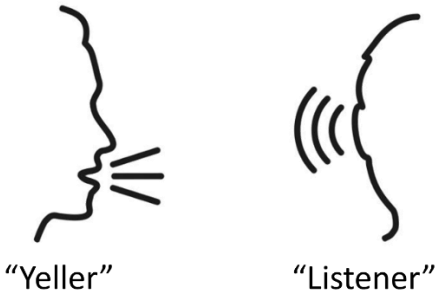
4.1.9 The Yeller Model (Simplification)

“DML models love to talk to each other”, I wrote above. But how do modules communicate with each other? There is talk of "flags" and "signals" and whatnot, but for something that is supposed to be simple to use, understanding how DML modules internally work with each other, should *not* be required. What we need is a metaphor, a figure of speech, an analogy that reflects the spirit, even if it may technically be somewhat incorrect:

(Please also see the demo mission “demo – The Yeller Model.miz”)

4.1.9.1 Of Old Yellers and El-Aurtians

Instead of getting down into the nitty gritty of techspeak, let’s try a more accessible approach. It holds up well and explains how this stuff works without needlessly getting into details. So, no "Flags" or other weird stuff. Let’s keep it simple.



So, how does, for example, a *unitZone* tell a *cloner* to produce new enemies? We can boil it all down to astoundingly simple things: "Yellers" and "Listeners".

When told to, "Yellers" shout out commands to anyone who would listen, and "Listeners" continuously listen to hear a specific command. If they hear that command, they get into action. They ignore everything else.

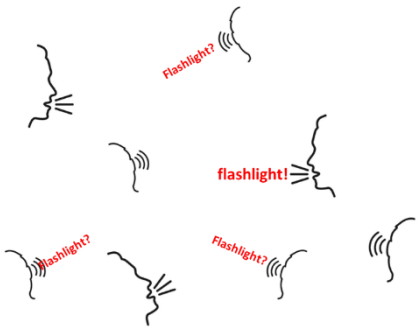
And here are the important points: Yellers don't care who listens to them, and Listeners don't care who shouts their command. Each do their own stuff, and they neither care nor know about the others. If Listeners hear a command other than the one that they are waiting for, they ignore it.

I call this "**DML's Yeller Model of co-operation**", or "**Yeller Model**" for short. Let's take a closer look:

4.1.9.2 A room full of people…

Imagine a dark room filled with Yellers and Listeners, all minding their own business. Now imagine that there are three Listeners in there who wait for the command "flashlight", and when they hear that command, they are instructed to turn on their flashlight for 20 seconds.

Now you also place a Yeller into the room and tell it to yell "flashlight" every 90 seconds.



What will happen? If everyone does as instructed, the Yeller shouts "flashlight" every 90 seconds. All Listeners hear that command, and the three of them who are waiting for "flashlight" shine their flashlight for 20 seconds.

So, every 90 seconds, the room is lit by three flashlights for 20 seconds. All the other Listeners in the room ignore the Yeller's command because it's not what they are instructed to wait for.

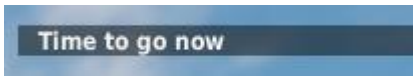
And that's basically how DML works. Some modules yell, others listen (some do both). A module with an "Output" (an attribute that ends on an exclamation point "!") can yell commands. For example, the *pulseFlag* module has an output "*pulse!*". The value that you put into the field next to it is the *command* that this pulser will yell - in our example it is the command "gogogo" and it is to be yelled every 90 seconds (*time* is set to 90)



A module with an "Input" (an attribute that ends on a question mark "?") listens for a command. If this input hears someone yelling that command, the module reacts.

For example, a *messenger* module listens on its "*messenger?*" input for the command "gogogo". It is set up to print the message "Time to go now" onto the screen when it hears the command "gogogo".

If you run that mission, the output "Time to go now" appears on the screen every 90 seconds.



In a nutshell: if one module wants to tell another module to do something, they use a common command or keyword. When it's time, the yeller blindly shouts its command into the open. So, yeller modules like *pulseZones*, *unitZones*, *radioMenus,* they all yell their commands, not caring who might listen.

The Listeners hear “their” commands only, and react by doing what they do: cloners clone, messengers put up a message, and smoke zones start or stop smoking.

So, if you set up an *LZ* module to yell “playerLanded” from its *landed!* output when a player lands inside it, and a *smokeZone* listens for the command “playerLanded” on its *startSmoke?* input, that smoke zone starts smoking when a player lands inside the LZ zone.



4.1.9.3 Of Inputs! and Outputs!

And that is how the “Yeller Model” works: “Listener” modules wait until they hear the correct command on their input and then do something. “Yeller” modules look for a certain situation to arise (for example 30 seconds have passed), and then yell a command into the void, not caring if anyone is listening.

And when you put these two groups together, you create a bigger whole. Crucial for Yellers and Listeners to work together is that they use the same command, and that there is no crosstalk (i.e. other modules that use the *same* command for different purposes. Logic inside DML prevents a module can’t hear a command because another module is shouting a different command at the same time – in DML, all commands are always perfectly received by all modules). Yellers shout their command through their *output!* into the blue, and don’t know nor care who receives their command; Listeners listen for commands at their *input?* and they respond to *any* Yeller’s command if it matches, no matter who yelled it. This makes it easy to build complex automatons that work across the entire map – and it makes it equally easy to goof if you by accident make a Listener wait for the wrong command or have a Yeller scream an already-used (existing) command across the mission, triggering unwanted action.

4.1.9.4 What’s in a name? Commands

There’s one thing that I haven’t mentioned yet, and that perhaps has you scratching your head: What are these ‘commands’ that I speak of, and where do I find them? You don’t. You make them up. Commands are ‘code phrases’ that you, the mission designer, assign to the inputs and outputs, and I recommend that you aren’t too creative with them. Use commands that are descriptive (like “SAM North destroyed”), and that help you to determine what they are used for. Moreover, make sure that you always spell them *exactly* the same. DML is case sensitive, and if you, for example, get its capitalization wrong by just a hair, a Listener won’t react to it. So be very, very careful how you name your commands, and mind your spelling and capitalization. You are (mostly) free otherwise, and can name your commands like you see fit. I only *recommend* that you do not use blanks in a command’s name, and use “CamelCase” instead: use “SAMNorthDestroyed” instead of "SAM North destroyed”. Oh, and **never use a comma** “,” in a command’s name.

Always remember that commands have to be matched pairs to work: there has to be at least one Yeller and at least one Listener that uses the same command. You *can* have three Listeners for the same command, no problem. And you can have multiple Yellers shouting the same command. But if there is no Yeller that shouts a particular command across the mission, your waiting Listeners do have a problem. Conversely, if your modules yell a command, and nobody is listening to that command, it won't have any effect.

4.1.9.5 Hiding the Details: it's a good thing!

The Yeller Model is a simplification of how DML modules work together; it's a nice and easy metaphor. The good part is that it works. The great part is that you probably won't need to know more - DML is set up to simply work this way.

Now, like all metaphors, they only work to a certain point before they break. Most mission creators can rest easy, knowing that they'll never reach that point. Occasionally, though, you may want to accomplish something trickier, something that requires that little bit of extra finesse. DML has you covered, but in return it requires that you know more about the underpinnings of the "Yeller Model": Flags and the incredible additional lifting power that DML provides should you want to get that really slick finish.

In this context, simply know that the Yeller Model's "commands" that modules yell and listen for are abstractions of a DCS concept called 'Flags'; that they can do much, *much* more. Simply nod wisely when someone mentions "flags", knowing that if someone uses that term with DCS, they in principle mean the commands that we talked about. Under the hood, they can be used for a lot more than what we do with the Yeller Model. Only few Missions require that kind of deeper knowledge and flexibility. And, if you *really* want to, DML provides you with a rich environment that manages DCS flags for you and can imbue them with additional properties to accomplish lesser magic.

As a result, if your mission requires some sorcery, you will need to resort and know flags. Until then, you should use the "Yeller model" and create great missions. Most of the missions that I have created work *entirely* with the Yeller Model. Yes, that includes "Expansion".

4.1.9.6 The Yeller Model in other parts of this Document

The "Yeller Model" is an easy simplification that I introduced to help people understand and more easily enjoy working with DML. The module description and all the demos are written with the (more technical) Flags model in mind. Translating that into the Yeller Model isn't difficult:

- Inputs and Outputs are the same in both models
- "Banging on a Flag", "sending a signal" or "receiving a signal", triggering all mean the same: yelling/hearing a command
- "Trigger" means sending/receiving a command
- DML Method (advanced stuff that wrap commands, ignore)
- Flags, Watchflags are synonymous with inputs/outputs (a *Watchflag* is an input)

Some concepts in the "Flags Model" don't translate to the Yeller Model and can be safely ignored. If the above translations don't help you understand what an attribute is for, you can probably ignore it.

4.1.10 The Flags model

Note:
This section takes a deep dive into “DCS Flags” and module-to-module communication. If you prefer The Yeller Model, feel free to skip this section, and come back if you need more detail.

DML modules can (and love to) talk to each other. In the last section, we introduced The Yeller Model, which explains how modules co-operate in a simplified way that should be sufficient to cover 99% of all mission design requirements.

The Yeller Model works well, but it’s a *metaphor*. Here’s your first stab at the truth beneath: Flags and the signals that they carry. Again, for most DML mission building, what follows below constitutes knowledge overkill.

4.1.10.1 Of Outputs! And Inputs?

When DML modules talk to each other, they pass information (“signals”) to each other and the sending module can trigger actions on the receiving module. One module can decide that it has detected something important, and then makes that information accessible on its output.

For example, the LZ module, as part of its duties, looks for aircraft that land inside its zones. When it detects that a plane has landed inside such a zone, it then sends a signal to its *output* for that zone. In DML, outputs are attributes, and their names end in an exclamation point “!”. If you look at a module’s description, you’ll find many such attributes – they are all outputs that will signal that something important (from the module’s perspective) has happened.

Just because some module can send signals does not mean that anyone can receive them. To receive a signal, modules must have the ability to listen. In DML, many modules “listen” for a signal on their inputs, and once they receive a signal, they react. The fireFX module, for example, will start its effects when if it receives a signal on their “start?” input, and end a running effect if it receives a signal on the “stop?” input. For visual clarity, and to mirror output attributes’ question mark, all inputs attribute names end in question marks “?”.

density	thickness or visibility of the smoke produced by this effect.
start?	DML watchflag (input) for when the effect should start. Defaults to <none>
stop?	DML watchflag (input) for then the effect should stop (Note: unlike the smoke zone module, which can take several minutes for the smoke to stop, this fireFX’s smoke stops within seconds) Defaults to <none>

So, if we connected the LZ’s output to a fireFX zone’s input, the LZ could cause the fireFX zone to start a fire effect when a plane lands inside the LZ’s trigger zone.

The upshot is that in DML, you connect outputs to inputs to build a greater whole, just like putting together a bunch of sensors and machinery in the correct way can create a lawn mower. DML teems with modules that specialize in looking at stuff and then provide convenient output signals for you: conquerable zones, radio menus, zones that detect units inside them. In classic toolboxes, these would be ‘detectors’: switches, sensors, dials etc.

They provide the means to produce output signal under strictly defined circumstances (e.g. “RED faction captures zone TankFactory”).

And DML is also chock full of modules that can do something for you should you tell them to: messengers, cloners, smoke/fire effects, flares, explosions. These modules all wait for a signal on their input to spring into action and do their stuff. When you connect their input to a sensor’s output, you – presto! – have a small automaton in your mission.

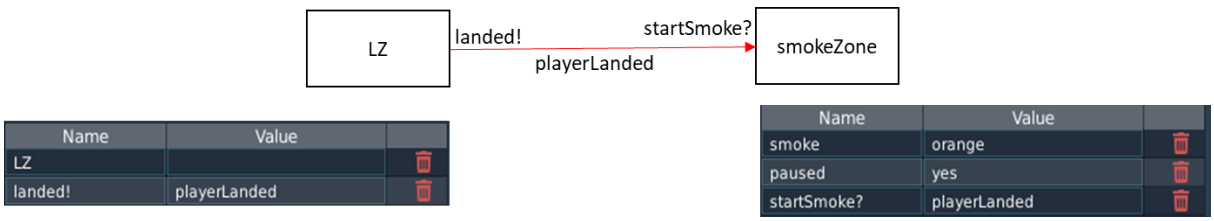
That’s DML.

4.1.10.2 Connecting the dots

To connect inputs and outputs, we use something abstract from the underlying DCS that it calls a “Flag”. What a Flag actually *is*, is quite arcane, and in the context of DML we can ignore most of it. What they *mean* to us is this:

- There is an endless amount of these “flags” available to you and each mission comes with its own set of flags, they do not get used up.
- They are here to serve you, the mission designer.
- All “flags” have names by which you tell them apart.
- You, the mission designer, get to name flags. Their name can be anything, and I recommend that you name flags in a manner that makes sense to *you*. I use flag names like “SenakiSAMSdestroyed”, and the choice is entirely yours A flag’s name has no significance for the flag itself, just to you.
- In DML, we’ll completely ignore what flags are internally. To us, they represent abstract, “phone lines” that you can tell apart by name and that connect outputs to inputs; they transport whatever an output wants to send to all the inputs that use the same phone line.
- You connect inputs to outputs simply by using the same *name* (i.e. phone line or “flag”) for both. So if one output! attribute has “startEngine” as value, all input? Attributes that have the same “startEngine” value receive anything that the output sends.
- Multiple inputs and outputs can share the same name (flag/phone line); just remember that all inputs listen to all outputs on the same ‘line’. So beware of crosstalk if there are too many senders (outputs) on the same line (a rare occasion in DML).

Using inputs and outputs can be reminiscent of old-style electronics and phone schematics, except that inputs only “listen” to the line, and outputs only ‘speak’ onto the line. This may seem initially but can be helpful: when designing a complex system, try to draw a block diagrams first:



4.1.10.3 Controlling modules through in/outputs

After all this talk of input and output, let's try to put this into some simple demonstration. There's an easy way to look at inputs and outputs: you use modules with an output to control other modules via their input.

In our demo we have a fire effect burning in front of out plane. Unimpressive, I know. Now let's do something that's easy to do in DML but takes lot more effort in vanilla ME: give the player control over the fire: we want the player to be able to switch the fire effect on and off via the communications menu.

The first step to enable this is simple: we already looked into fireFX's documentation and know that it has two inputs that control the effect:

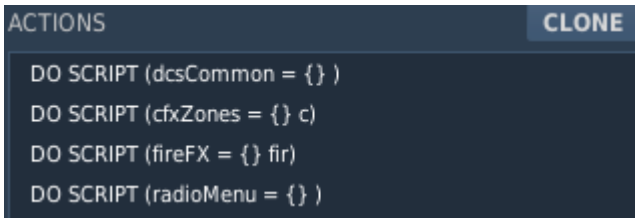
- *start?* a signal received on this input will turn the effect on
- *stop?* a signal received on this input will turn the effect off

All we need to do is add these two attributes to the "You're Fired" trigger zone, and decide upon the name of the flags ('phone lines') that they will use to receive a signal. Arbitrarily, I picked the names "flameON" for *start?* and "extinguish" for *stop?*, simply because I like it when flag names tell me what they are used for.

Name	Value	
fireFX	small	
onStart	yes	
start?	flameON	
stop?	extinguish	

So now we have a fire effect hosted inside trigger zone *"You're fired"* that sports a *"small"* fire effect (with *flames*, implicitly via default) that starts when the mission starts (*onStart* = *true*) and can be turned on and off at will through signals on the flag named *flameOn* (connected to *start?*) and *extinguish* (connected to *stop?*).

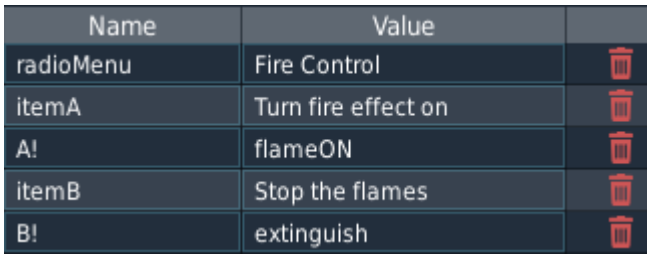
All we need now is a convenient way to create these signals. And for this we turn to one of DML's "to die for" modules: radioMenu. If you have created some missions, you probably know that DCS provides mission designers with some methods to add interactive menus to the "Communications→Other" tree, accessible in the cockpit. DML provides its own, massively improved way to do the same – only better, simpler, and faster. Yeah, and I'm not even bragging here. The radioMenu module has pretty much only one purpose from a mission designer's perspective: to provide output signals. The fact that these are controlled by player is just the icing on the cake. So the next step is to add the radioMenu module to your mission. Go ahead, you know what to do. The resulting AT START action list should look similar to



Now, being one of DCS' "Crown Jewels", it has more features than you can shake a mouse at, so we'll condense everything to the outputs that we are looking for, and how we can get them to work. First, we create a new trigger zone and call it 'player menu'



and tell it to host one instance of this module by adding a “radioMenu” attribute with a value of “Fire Control”. The latter appears in the player’s Communications→Other menu as its own entry.



Since we know that output attributes end in “!”, they are easy to find in the documentation: radioMenu can provide up to four outputs per zone, their attributes labeled “A!” through “D!”. We’ll use “A!” to turn on the fire by using the value “flameON”, and “B!” to turn it off by connecting it to “extinguish”. We also provide some narrative for A! and B! through the attributes “itemA” and “itemB”, completing the entire radio menu trigger zone.

We have built our first user-controlled effect in DML:



Let’s try this in-game. Run the mission, and hit the Communications key for your aircraft. Press F10 to access the “Other” menu and explore the options. Note howthe menu structure relates to what we entered into the attributes for the radioMenu:



In the “Other” menu, notice the *“Fire Control”* entry. This is the menu that the *“player menu”* zone installed (see the *“radioMenu”* attribute’s value, above). Choose *“Fire Control”*, and the next level of menu items come up. We get to choose between *“Turn fire effect on”* and *“Stop the flames”*, which are the values we entered for the ‘narrative’ attributes itemA and itemB.

Choose “Stop the flames”. The flames peter out; imperceptibly at first, but after some 10 seconds flames and smoke should be gone.

Choose Communications→Other→Fire Control→Turn fire effect on and see the ground erupt in flames and smoke once more. We are using the radioMenu module in the player menu zone to control the fireFX module that is hosted by the “You’re Fired” zone. The

radioMenu tells the fireFX to start via the *"flameON"* flag that connects the *A!* attribute output to the *start?* input. Likewise, the flag named *"extinguish"* is used to connect radioMenu's output *B!* to fireFX's input *"stop?"*

And if all of this makes your head spin - don't worry, it will quickly become second nature.

4.1.11 What? Where? When? How?

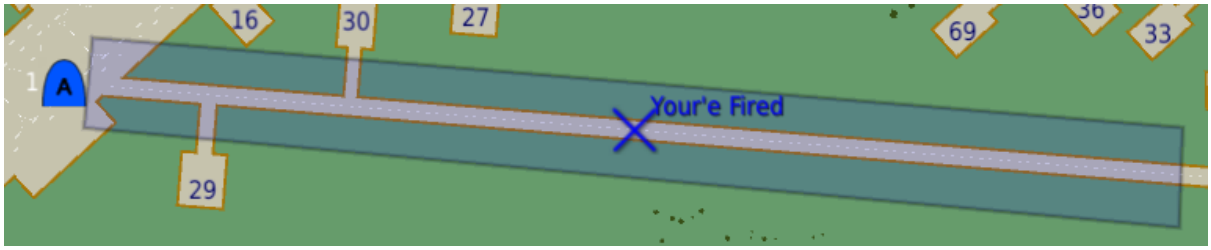
After seeing a bit of what modules can do, let's try and pull these things together to get a better understanding how the various parts work together, and then have some fun (after all, we have a fire effects module to play with that we can trigger at will from our cockpit. Let's have some *fun*!)

DML uses trigger zones and interacts with them so it can achieve remarkable things that you can use in your missions. So how do DML and DCS work together? Big picture view:

- *Trigger zones* tell DML *where* something is happening. Many modules can use the trigger zone information (location, area, boundaries) to focus their effect on. ownedZones, for example, treat the surface area of the trigger zone that it is attached to as the area as the area that the factions fight over. It regularly checks who (which faction) owns the units that are inside the boundaries of the zone, and then evaluates who the owner is
- *Module Key* Attributes tell DML which of the modules that you have added to your mission should attach to this trigger zone. For example, the 'fireFX' attribute tells DML that a zone with this attribute is to host the fireFX module. Note that multiple zones can (and often do) host the same module: a mission can have multiple zones that host cloners or fireFX. Since a module defines what happens, the key attribute decides what happens inside the zone.
- *input?* attributes usually control *when* something happens in a zone. In the "Yeller Model" this is the "command" that it listens for to trigger the action.
  Most modules are busy doing nothing for 90% of the time or more. Then, when they receive a signal on one of their inputs ("hear their command"), they spring into action. Cloners, for example, sit around and wait until a signal ("command") tells them to create more clones.
- *output!* attributes usually control other modules and tell them to start or stop doing their thing – they control when something happens in other zones. This is what yells a command in the "Yeller Model".
- *many other attributes* change how something happens inside the zone, how the hosted module reacts to inputs, processes information, selects units, and generally goes about its business.

It's the *many other attributes* part that makes DML so versatile and is one of the main drivers of DML's continued development. Let's play with our current setup of the fireFX and change some of the 'other attributes' to get a better feeling of what it can do.

First, let's change the area of effect: change the trigger zone that hosts the fireFX module: make it a quad-based zone, and enlarge it. A lot:



Run the mission. As expected, you'll see the fire erupt exactly where you see the trigger zone's center in Mission Editor.

So let's get some random involved: fireFX has the ability to randomize the location where it places the fire inside the trigger zone. This is controlled by the attribute *"rndLoc"*. Add the attribute, set it to *true*, and run the mission a few times. Note that each time that the mission starts, the fire is placed differently, and always inside the trigger zone "You're Fired".

Name	Value
fireFX	small
onStart	yes
start?	flameON
stop?	extinguish
rndLoc	yes

This means that if we wanted to place some random fire over the airport's location, we could now copy this trigger zone, and paste it a couple of times over itself, and we'd have that many fires placed at random locations inside the zone.

Or, we could do it the DML way: there's a *num* attribute that controls how many fires are to be generated inside the trigger zone. And since it's a common requirement for mission designers to also make the number of fires as random as their location, DML's fireFX zones also support that: you can tell it that you want the number of fires should be in a range "from-to" – and DML will do the rest. Let's set *num* to "10-20"

Name	Value
fireFX	small
onStart	yes
start?	flameON
stop?	extinguish
rndLoc	yes
num	10-20

And this is how one of the results can look like



4.1.12 So what?

Agreed, these are near-trivial examples, and some of you may shrug and say "so what?"The big thing with DML is that you can use trigger zones and their attributes to

- to place new abilities or better control existing abilities
- to change the way abilities behave
- to connect abilities

and never have to change a single line in some script. DML gives you access to all its abilities and ties them together for you transparently. All you have to do is design great missions. DML will manage the tedious stuff in the background for you.

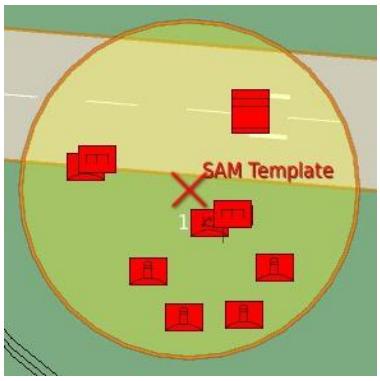
4.2 DML “Crown Jewels”

Over time, DML has grown. What once grew out of my desire for a small framework for some dynamic missions has now become a large, powerful toolbox for wide plethora of mission types that also happen to include dynamic missions. While DML grew in power, it also picked up a lot of nice-to-have features. As it stands now, DML encompasses more than 40 different “building blocks” and 20 “Big Ticket” items – a bewildering selection of modules to choose from, and that holds at least a couple of great module for anyone’s taste

So if there were only a few modules that I’d deem “too good to miss”, which would that be? Here is my selection of DML’s “crown jewels”, modules that are in nearly every mission that I create:

4.2.1 cloneZones

This is hands-down DML’s biggest over-achiever. It solves more challenges than you even knew you had. CloneZones spawn units on-demand. CloneZone’s big twist is that is uses “templates” for the groups that it spawns, and you, the mission designer can design in Mission Editor the group composition and route that they are to take. Not just ground units, by the way. Of course cloners will happily clone your flights and naval strike forces. And cargos. And static objects. And all within Mission Editor, for comfortable editing.



And that’s just where the cloners begin, they can do a lot more: randomize spawn locations, re-spawn when the last unit was destroyed, place cloned units on roads – even when their location is randomized. Oh yes, and of course cloners can access templates randomly.

So, cloners can pretty much cover any ‘dynamic’ requirement that you have, and people have told me that they prefer cloners to “normal” ME late spawning (via inactive groups) because it’s so much easier to simply send a signal to a cloner than synchronize time tables.

Ah, yes, before I forget – since cloners are DML building blocks, they can be attached to units. So you can have a cloner that can dynamically spawn units while it follows around another unit… The next time someone asks how you’d create a group of soldiers that jump of a truck when it’s attacked, you now know a good answer.

4.2.2 messenger

Mission designers’ options to place text in front of their players’ eyes are decidedly limited. We have MESSAGE TO. And – outside scripting – that’s pretty much it. Worse, the text that mission designers can put in front of players is entirely static. There is no way to tell a player how many minutes they have left unless you have fully authored that message.

Messenger changes all that with its deep, comprehensive method of “wildcard text”. The that functions is that when you author a mission, you create ‘boilerplate’ text that contains wildcards. During playtime, these wildcards are converted dynamically to meaningful text. And that changes everything, as wildcards are phenomenally flexible, and can even include ranges of words that you prepared for messenger to pick from. It can be location information

(in lat/lon or MGRS), bearing, distance, time, and many other things that I have never imagined but people show me they managed to create with messenger.

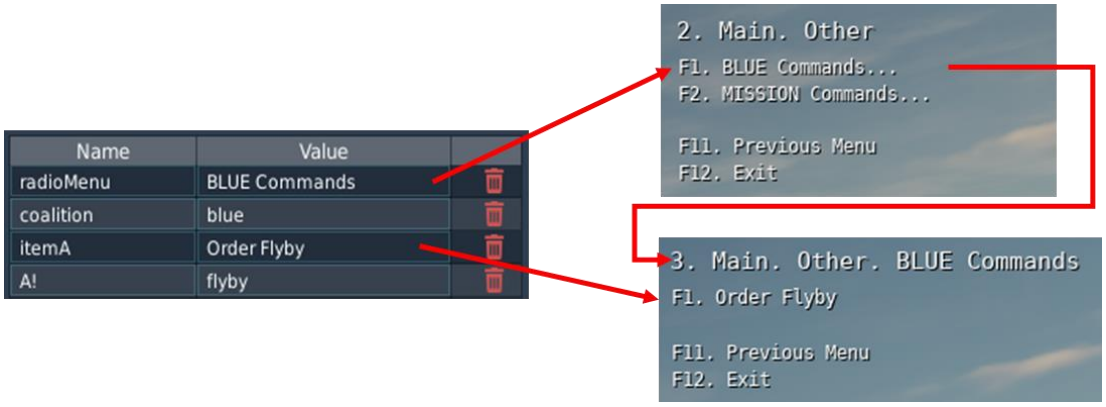


Every mission needs to talk to their players, and when it comes to do that, messenger is a pretty good choice.

4.2.3 radioMenu

You've already met radioMenu – and that's no coincidence since it's the best tool in your box to create user-interaction. User interaction is what makes or breaks missions, and having ready access to menus makes authoring engaging missions so much simpler.

That much is obvious. The not so obvious part is that – especially with signal-driven DML-type mission authoring, radioMenu is your low-key debugging tool. I often use radioMenu when I'm testing missions to selectively advance mission stages. It creates signals on-demand, and outside of THE DEBUGGER nothing can match that ability and convenience.



4.2.4 heloTroops

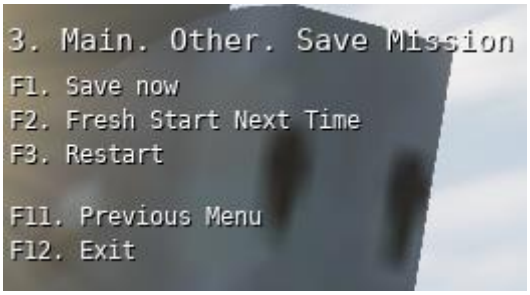
This "big ticket" drop-in module adds a significant player ability to your missions: use helicopters to load, transports and deploy infantry. Since it's a drop-in module, all you need to do is add the module, and your mission now allows players to transport infantry. This allows capture mechanics, or any other troop insertion/extraction idea that you may come up with.

4.3 DML “Dark Horses”

DML’s “Crown Jewels” are obvious - anyone who has spent some time with DML would probably agree. There are more, somewhat obscure, and less sparkling gems that I think you should at least be aware of.

4.3.1 Persistence

DML can save your mission to storage, and then “restart” the mission from file. The geek term for this process is “persistence”, and DML’s persistence is fully fledged.



Currently, however, it comes with a lot of baggage due to the way DCS is put together, not to mention some security concerns. So, should you ever wonder if you could build a mission with DML that can save and resume, the answer is a qualified yes. Yes it can save. Yeah, it can kind of resume. You don’t have to push the issue now, just remember that yes, there’s persistence built into DML.

4.3.2 ssbClient / slotty

This “Big Ticket” combo provides a simple, but inexplicably absent feature to your missions: blocking player slots on airfields that are owned by the opposition. By adding *ssbClient* (provides the logic which slots to block) and *slotty* (provides the actual slot blocking) modules, capturing airfields becomes a strategic imperative rather than option.

Note that there is an optional server module called “SSB” that (on multiplayer missions) provides a stylistically more satisfying (and better UX for) slot blocking when installed, and that is 100% compatible with *slotty*.

4.3.3 THE DEBUGGER

The darkest gems of all, a black diamond. Few people know it exists, fewer even have an inkling of the raw power it harnesses for you when you embark on debugging your missions. If you leave the sunny meadows of plain mission design and descend into the murky depths of multiplayer, dynamic persistent missions, make sure that you at last know about the big gun that you can call in. THE DEBUGGER’s abilities far exceed DML’s requirements and can routinely assist script-heads when they test mission code (small wonder, it helps me develop DML). But above its neutron-star hard-core level support for the unspeakable, it features very helpful ME integration, can track events for you, and of course will diligently watch flags for you, note when they change and tell you about it.



PART III:

REFERENCE

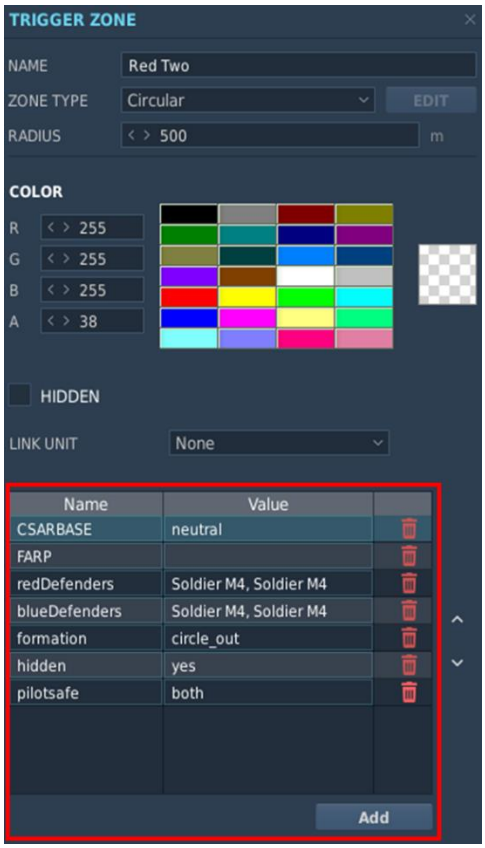
5 DML Reference

5.1 DML Core Concepts

DML uses several easy-to-understand concepts that it uses to meld advanced mission abilities with classic mission design in Mission Editor. In the remainder of this section, we are going to discuss how these concepts work throughout DML – they are shared across modules, understanding them will greatly facilitate your mission design efforts

5.1.1 Zones and Attributes

DML uses an existing, central DCS Mission Editor tool to integrate into your workflow: Trigger Zones. They can be placed anywhere on the map, are easy to modify (move, change, copy and paste), and they already support a central feature that we now use to pass information from ME to our DML modules: *Attributes*.



Attributes are called “named values” (or “properties” or “name/value pairs”) in literature. Mission designers can freely add, modify, and remove attributes from Trigger Zones. An attribute always has a name, and a value. And you can easily use ME to change both.

Module “Anchors”

So now that we know what attributes are, why are they important for DML? They help DML to know where you want to put which module to work: it scans all trigger zones, and in each trigger zone, it looks for attributes with certain names (e.g., “smoke”, or “fireFX”), and if it finds an attribute with that name, it automatically “anchors” the appropriate module (a smokeZone or fireFX module) to that zone. For example, DML connects the smoke zone module to any zone that has an attribute named “smoke”. Read the “Reference” section to find out which module looks for which attribute. It usually is the attribute that is also deemed “MANDATORY”.

The image above on the right shows a Trigger Zone called “Red Two”. In the lower part a red box highlights the attributes that were added to this trigger zone. You can add as many attributes to a zone as you like, if a module doesn’t recognize an attribute’s name, it simply ignores it. All DML modules use Trigger Zones with attributes to anchor their modules, and to control a module’s functionality.

Usually, the name you give to a Zone itself (“Red Two”) is irrelevant; DML looks for specifically named attributes to anchor modules. You can also use the same zone to anchor multiple modules – the example above **anchors three modules**: CSABASE (which anchors csarManager to this zone, allowing CSAR Missions to complete here), FARP (which anchors

FARPzones to this module, which manages red/blue defenders and supply for the FARP when owned by a faction) and PILOTSAFE (which anchors limitedAirframes here and allows pilots to safely exit their aircraft and change into new ones).

5.1.2 So what, exactly, are Attributes?

For our mission purposes, an attribute is something that has a name (for example 'flames) and that has (or needs) a value (for example "true"). When you place a trigger zone and attach a module to it, DML automatically adds all attributes with all default values so that everything works out of the box. You can then override those defaults simply by naming the attribute's name (for example "flames") and the value that you prefer (for example "false").

Not all attributes can have just any value - after some time, you will see some patterns emerge, as DML knows only a small variety (or different 'kinds') of attribute values:

- *Numeric Values*
  Modules use this when they expect a number, for example the time delay for a fuse, or the number of fires that should be placed in a zone. Examples:
  - *24*
  - *-100*
  - *3.14*

- *Ranges*
  A range is a special case where a module can accept not just a single number, but a 'from-to' numerical range like from 1 to 100. A range are two positive whole numbers that are separated by a hyphen (or "minus-sign") "-". They are often used to randomize a modules effect, like a time delay or number of enemies to spawn. A DML module can only recognize a ranges when and where it explicitly tells you in the description that it does. Examples:
  - `1-100`
  - `2-7`

- *Text Values*
  This is the most common kind of attribute, and it is used for messages, flag names, and similar. Examples:
  - `Hello Pilot, welcome to your Cockpit`
  - `You are leaving <z>, good Luck!`
  - `playerDeparted`

- *Lists*
  This is a special variety of the Text Value, and DML can only recognize this if it tells you that it also expects lists. Lists are Text fragments that are separated by a comma ",". This is most often used to prove multiple values – for example flag names, liveries, or types. A list may sometimes support ranges, but only if the module's description explicitly says so. Examples:
  - `A-10A, F-15C, UH-1H`
  - `f1, f3, ab, 77-82, T3`

- *On or Off – and "maybe": "Boolean-ish*
  Many attributes control if a feature is enabled ("on") or disabled ("off"). In geek speak these are called 'Boolean', after some mathematician who lived in the 1800's and

thought so much about the concept of things that can be either-or, that people named the whole after him: George Bool. The values most often associated with this are “true” and “false”, and are of immense significance inside the computer world.

In DML, we give this concept a small boost, because instead of “On” and “Off”, you can also assign a third value: “maybe”.
Now, most of the time we will use one of the two classic values: On or Off (or their synonyms). Sometimes, however, we don’t know because we want the mission pick a random value when it starts up. In these cases, the mission picks a value by random, and then sticks with that value throughout the rest of the mission. This is most useful for attributes that control something at mission start, and when you want to introduce an element of surprise. For example, the cloneZone’s *onStart* attribute can be either on or off, and if you supply “maybe” as its value, that value is chosen randomly when the mission starts.

When DML expects a value that is true or false, you can opt for the following values:

- **TRUE**: use any of the following (not case sensitive)
  `True`, `Yes`, `1`, `On`

- **MAYBE**: (resolved to true or false at mission start with 50:50 probability)
  `Maybe`, `RND`, `Random`, `?`

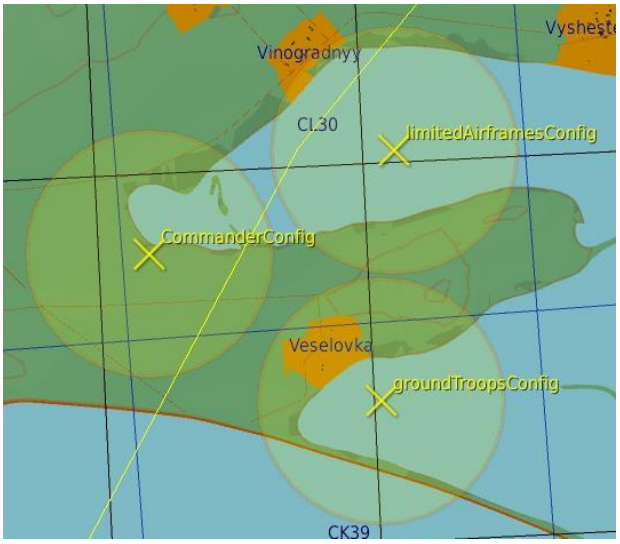
- **FALSE**:
  Anything else

5.1.3 Module Configuration Zones

There *are* cases where a zone’s name can be relevant: the most common case is to help to (globally) (re-)configure an entire module: DML uses so-called “Module Configuration Zones” (see below) that can be omitted entirely (if you do not want to curtail the way that a module works); that you can place anywhere on the map, and that *must* have a specific name in order to be recognized. When a module can change the way it works with a config zone, the module reference will tell you exactly how to use it; using config zones is completely optional, and very simple:

Since Trigger Zones are so convenient to pass data from the mission to modules, modules also use them to provide data for global settings (“configuration values”), and to pass data for processing. For example, the *civAir* module uses a config zone to set the maximum number of civil aircraft at any time, and a data zone to control liveries and aircraft types for civilian flight. Both are optional, and modules run well without config zones; they simply use default values, and when you use a config zone, you only need to add attributes for those values that you want to change.

Like before, Configuration Zones (and data zones) use attributes with values to pass data to the module. Unlike before, 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. Since they are global, it does not matter where you place them. I usually reserve a remote location on the map for them, where they are out of the way, yet easily accessible.



In above example, we see three configuration zones: one each for commander ("CommanderConfig"), limitedAirframes ("limitedAirframesConfig"), and groundTroops ("groundTroopsConfig"). As mentioned, for configuration zones and data zones, their **trigger zone name** as given in Mission editor *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's behavior 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.

As before, config zones make strong use of defaulting – even if you think that you need to resort to a config zone, you'd usually only include the one or two attributes that you intend to configure differently from their default. This makes configZones easy to maintain.

And again – the placement (location) of a configuration zone irrelevant, you can place them anywhere you like. A good place for them is somewhere out of the way where they can't be confused with, or get in the way of other mission-relevant "action" zones (one of the corners or edges of the map, for example). Some people also like to color-code config zones (I often use yellow – there is no "official DML color coding" for zone, use whatever you prefer).

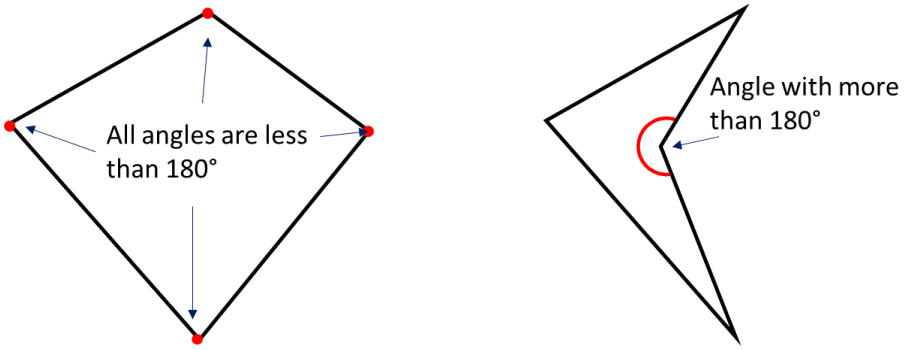
5.1.4 Zone Shape

DCS supports two different types of zone shapes: circular, and 'Quad-Point'. The latter is a zone that is defined by four corner points and can be used to define irregularly-shaped zones.



DML supports both zone versions with a small restriction: When you define a quad-point based zone (also called ‘polygonal’ shape, ‘many-edged’ because it has, uh, many edges) and use its shape to determine if something inside that zone, the zone’s shape must be ‘convex’. While this sounds terribly technical, it basically means that if the zone’s shape looks like a ‘V’, it is **not** convex. The result is that for such a non-convex shape, some of the ‘inside’ calculations may be off. Now, the good news is that for mission design, *almost all* quad-point based zones *are* convex shapes, so you will rarely encounter this restriction. If you have issues with a zone not correctly detecting if a unit is inside, check if the zone itself is V-shaped. If so, try and resolve this by creating two triangular-shaped zones instead.

The ’formal’ definition of a convex polygon (or quad-pointed shape) is that all inside angles are 180° or less.



But why does DML have such an obscure and somewhat arcane restriction that only applies to a tiny fraction of all cases anyway? Because the code that is used in DML to calculate if a unit is inside, and the code used to create a random point inside the zone is very fast, robust, and relies on simple geometric rules that can also work with more complex zones than those marked out with merely four-points – DML already supports polygonal shapes with an arbitrary number of points. Well, mostly: since ME doesn’t support it, that’s only theoretically, but DML is future-proof in this regard.

5.1.5 Flags in DML

Under the hood, DML uses flags to build missions – just like builders use brick and mortar to build houses: everywhere, for almost everything. But that does not mean that *you, the architect* of the mission, must know *why* they work as long as you know *how* – as long as you know what is possible and what isn’t. With that being said, it can help to better understand flags when your missions reach higher levels of complexity.

This section provides some deeper information about module inputs, outputs and the flags that they use to communicate with each other. Initially, you should skim this part, and return if something has piqued your interest, or you have run into a specific issue that needs explaining. If you have read and understood “The Basics” you already know enough to get started on building missions.

What follows is intended for the eternally curious – and those of us who simply demand to understand *why* something works, not just how.

5.1.5.1 Introduction

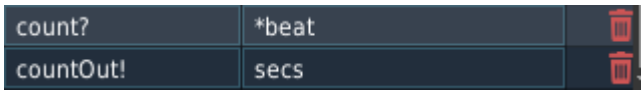
Outside of ME Trigger Zones, there is another central Mission Editor element that a mission uses to shape the flow of control: ‘Flags’. Missions use Flags to remember game states. For

example, if a SAM site has been alerted, a mission can use a flag to store that information so it can retrieve it at a later point or pass that information to another part of the mission. Although engineering-wise Flags in Mission Editor are primitive, they *can* be (and are) used to great effect – as many existing missions show.

DML's modules heavily use flags to communicate internally. Most importantly (and unlike DCS itself), though, **DML does *not* require a deeper technical understanding of flags** from the mission designer. It usually suffices to picture them as 'named wires' (like telephone lines) that connect outputs to inputs, and that modules use these 'wires' to exchange information. When it comes to flags in DML, the most important task for mission designers is to remember the names that they gave these wires, and to not mix them up.

Therefore, when you use a module, it may request a flag name from you that it connects to an input: it then gets information from that flag into the module. Or it may want a flag name that it connects to an output so it can put information onto that flag. It's up to you, the mission designer, to manage and provide the flag names. You usually do **not** need to concern yourself with *what* or *how* information is passed between modules. DML handles that for you transparently. What *is* important is that you remember which flag names you hand to DML (so you do not accidentally re-use a flag name and 'cross wires'), and that you spell those names correctly (so you do not accidentally 'break' a connection between input and output because the flag names do not match).

Flag names are given to a module in trigger zone attributes. For convenience, DML modules signal what they are used to in their *attribute name*:



- **Inputs: end on "?"**
  when a module looks for **input** (which often triggers some action) that **attribute's name ends in a question mark "?"**. For example, the cloneZone waits for a signal to create clones. It has an attribute named '*clone?*'. You pass the name of the flag that the clone zone should watch for a signal as value for that attribute. Module inputs in DML are designed to be fed from module outputs (see below) and work seamlessly.

- **Outputs end on "!"**
  When a module wants to pass information to other modules, it needs a flag that connects to an output. Attributes that are used as outputs have an exclamation point "!" in their name, and you pass the name of the flag that the output should connect to as value for that output attribute. For example, a unitZone passes its signals to the flag that you supply as value to the '*enterZone!'* attribute. **Output attribute names end on an exclamation point "!"**. They are designed to seamlessly feed into module inputs to trigger actions.

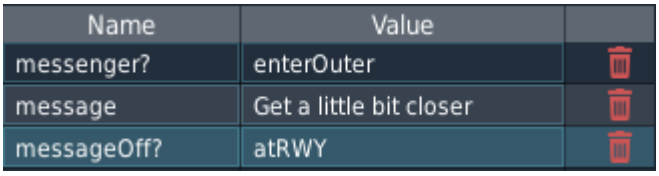
- **Real-time/changing "direct" value outputs "…#"**
  Some modules keep track of stuff for you. They have the ability to pass a "live" (or "direct") number to other modules – like the number of kills a faction has accumulated, or the number of groups that a module is tracking. These modules have special 'direct value attributes' that passes this 'raw' (unprocessed) value to flags, similar to how output attributes put their signals into flags. **When an attribute's name ends in a hash mark "#", that attribute is a 'direct value' attribute**. The difference is subtle: "outputs!" send a processed signal (in form of a

method, see below) via named flags when something significant for the module happens. “direct#” regularly (once every second) put a raw value into a named flag, irrespective if that information is relevant of not.
For example, the playerZone counts how many players are inside a zone. It can put a direct count of the numer of players in the zone into the named flag that you connect to the ‘pNum#’ attribute. This direct value is often helpful – but beware: just because the number did not change doesn’t mean that nothing happened in the zone. If within the same second a player left and another player joined the zone, the total (direct) value would be the same. The playerZone module would not have missed this – the (processed) outputs added! and removed! will both have sent signals, while the direct output looks as if nothing has changed.

5.1.5.2 Connecting Flags to Modules

Periodically (usually once a second), modules in DML check the flags that are connected to their inputs. If the value of the flag is different than the last time that the module checked, an action inside the module is triggered. This, in a nutshell, is how inputs work.

When a module deems it appropriate (usually when it thinks it has something important to share), it changes the value of the flag that is connected to its output. For example, a zone that this module watches has been conquered. The module then reads the current value of its output flag, changes that value, and then puts the changed value into the flag. Any module whose input flag connects to that same flag detects the change, and reacs.



As described before, you tell a module which flags it should use for input or output purposes. For example, the ‘Messenger’ module is triggered via the input *messenger?* to display a message. Another trigger (*messageOff?*) can tell this messenger module to turn off (mute it). Since triggering is done by a flag’s value, you decide which flag (identified by name) to use for which purpose. To connect a flag to an input, you add the flag’s name as argument for the module’s input attribute. In our example, the flag named “enterOuter” connects to the *messenger?* input, and the flag named “atRWY” connects to the *messageOff?* input. The result is that every time that the value in the flag named “enterOuter” changes, the messenger is triggered and displays the message “Get a little bit closer”. And when the value of the flag named “atRWY” changes, the “messageOff?” input is triggered, which turns the messenger off.

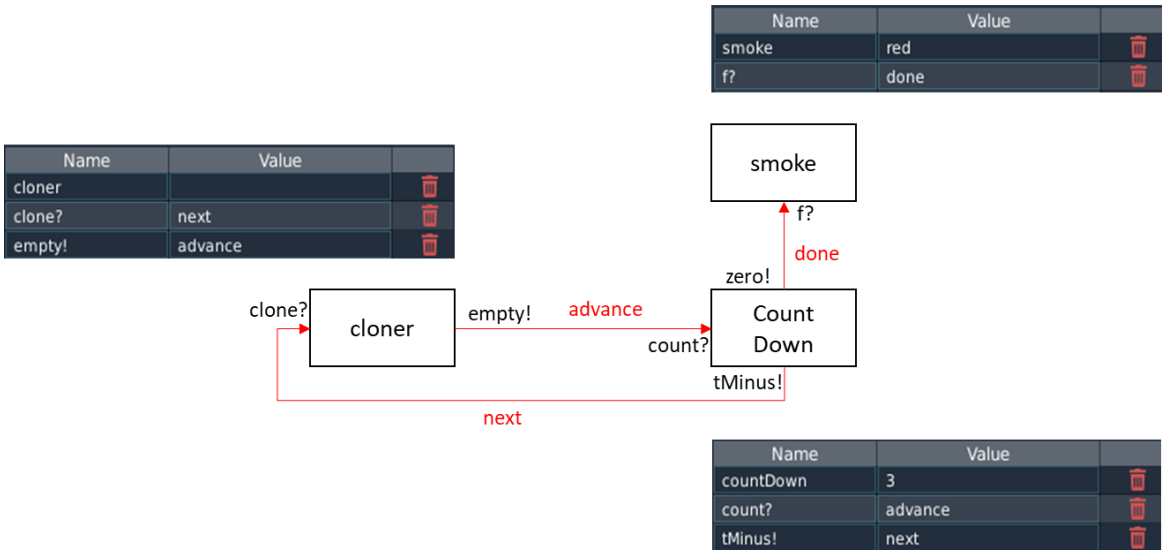
Conversely, a module can determine that it has identified an important situation and needs to signal this: for example, a conquerable zone has been conquered, an object destruct detector has detected the destruction of the object it watches, or a cargo delivery zone needs to alert the mission that a helicopter has successfully delivered cargo inside it. Whenever something like this happens, these modules generate signals on their outputs, and change the values of the flags that connect to them.



In short, modules generate signals to drive and co-ordinate mission flow. For example, a pulser sends out a repeating signal for other zones to synchronize their actions. In the above example, the pulser’s output *pulse!* is connected to the flag named “fire”: now, each time that the pulser sends a signal, it changes the value of the flag

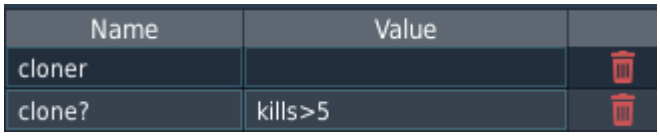
named “fire”. Other modules that connect their “inputs?” to the flag named “fire” will be triggered each time that the pulser sends a signal.

Anyone who has spent some time with digital circuits senses DML’s kinship with digital circuits of days past, and it can be helpful for some to envision the flow of control though flags similarly:



The image above (taken from the “once, twice, three times a maybe” demo, see that chapter in this document for an in-depth analysis) illustrates how three modules (a cloneZone, a smokeZone and a countDown) are ‘wired’ together to produce a ‘circuit’ that spawns enemies exactly three times; after the third time, it stops spawning and turns on red smoke to indicate that the exercise is over. Note how the **red named flags** connect the different module’s “inputs?” and “outputs!” to create a “logic circuit”.

5.1.5.3 DML “Watchflags”



You want to trigger a module’s ability (e.g. spawn a group, start smoke, output a message) at precise moments in your mission. To tell a module to activate we use flags that connect to “inputs?”. DML is built in a way that the various “inputs?” and “outputs?” connect seamlessly and work out-of-the-box. The default way how modules talk to each other is via named flags, and “outputs!” change a flag’s value, which an “input?” detects and reacts to. This works for 99.9% of all mission requirements. But DML wouldn’t be DML if it was content with that. Skip the remainder of this section until your mission’s complexity grows beyond DML’s defaults.

Because DML allows mission designers great flexibility when it comes to deciding how a module should trigger that goes far beyond simply triggering on a change in value. Advanced users can tell DML exactly what signal to look for. For example, a mission designer can tell a cloner only to spawn new clones when the value of the flag named “kills” is greater than 5. Let’s see how to do that.

5.1.5.4 Input Methods



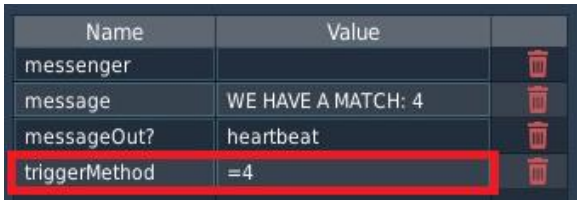
As discussed before, attribute names that end in a question mark (e.g., “f?”) are *inputs*, which connect to a flag. You supply the flag’s name (here “startAction”). All flags in DCS have alphanumeric names like “EnemySighted” or even old-style numbers like “100”. Every named flag always has a value - a number that can change as the mission progresses. In DML, we also call a flag that an input is watching a “**Watchflag**”. Because, uh, that’s obvious.

By default, if the value of a watched flag changes, that triggers a connected module’s input and causes the module to do its stuff - for example, start some colored smoke inside the trigger zone it is attached to.

In addition to triggering on a value change, DML optionally supports a host of other (additional) conditions that you can apply to watched flags. In other words, a module can be more selective about when it reacts to an input (e.g., only trigger when the flag’s value changes, *and* the new value equals a certain number). As mentioned before, an input’s *default* behavior is to trigger when the value of the flag that is connected to the input changes. But DML can do a lot more. Let’s look at the other options that you have:

Trigger Methods

After you tell a module’s input (the attribute ending in a question mark ‘?’) *which* flag to watch, you can also tell it *what* to look for. For most mission designs, the default that DML provides (look for a change in the flag’s value) is enough. This conveniently reduces complexity for mission design. Let’s open the decision box, and take a closer look: For a module to recognize that it should trigger on an input the following must be true



1. The associated flag’s **value must be different from the last time that it checked**. DML Watchflags only trigger on flag value changes and ignore the flag otherwise
2. The new value of the watched flag must match the rule (condition) that is described in the module’s **trigger method** attribute. By default, DML sets this method to “change” – whenever the new value is different from the last time it checked, that is sufficient: it fulfils the trigger condition. There are other conditions that you can choose from: In the example above, the input “*messageOut?*” that is connected to the flag “heartbeat” only triggers if the **value** of the flag “heartbeat” **has changed**, **and** the new value of that flag is equal to the number 4 (“*triggerMethod*” is “**=4**”).

In DML, we call the conditions that a flag’s change must fulfil before they trigger a module’s activities “**trigger methods**”, or just “**methods**”. So, unless you set your own *triggerMethod*, DML sets it to “change” for you: simply changing the watched flag’s value is enough to trigger the module’s function. In other words: when you connect a module’s input to a flag and don’t tell DML what to look for, it triggers whenever that flag’s value changes.

This greatly simplifies mission design with DML: detecting a ‘change’ in a flag’s value suffices for most of your mission demands and is the preferred method when you connect modules to each other. DML modules by default produce a flag value change on their outputs (see “*Bang! method (output)*”, below), which fits nicely with inputs. And it also conveniently integrates into classic mission design:

For (an old-school) example, let's say that if all enemy tanks are destroyed, your mission changes a flag through an ACTION: FLAG INCREASE that you set up in ME's Rule Editor:



If that flag is wired into a cloner's "clone?" input (*without* adding a triggerMethod to that cloner), the change in the flag's value triggers a clone cycle in the cloner. It's that easy, DML usually works well with your older designs.

Sometimes, though, you want finer control when something happens, and this is where the trigger "methods" come in. These "methods" are additional rules ("conditions") that are applied to the input flag's value. For example, the rule "=4" means that in order to to trigger, the connected flag's value must not only change, but after the change the flag's new value must also be equal to the number 4.

DML supports a host of input flag conditions, they even allow you to compare the flag in question to other flags. DML supports the following "methods":

- `'change'` or `'#'`
  trigger whenever the watched flag's value changes. No other conditions need be fulfilled. **This is the default**

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

- `'on'` or `'1'` or `'yes'` or `'true'`
  triggers when the watched flag's value changes from zero to non-zero
  (**Warning**: with this method, 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. To trigger again, the flag must first return to a value of zero)

- `'inc'` or `'+1'`
  triggers when the watched flag's current value is greater than the previous value

- `'dec'` or `'-1'`
  triggers when the watched flag's value is less than the watched flag's previous value

- `'lohi'`
  triggers when the watched flag's previous value was zero (0) or less and the new value is greater than zero. Often used with pulses.

- `'hilo'`
  triggers when the watched flag's previous value was greater than zero (0) and the new value is zero or less. Often used to detect a countdown reaching zero.

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

  Examples:
  - `>4` triggers when the watched flag's value is larger than the number 4

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

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

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

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

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

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

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

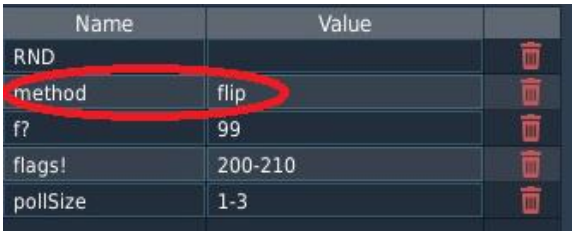
Quoting Numbered Flags
Early versions of DCS used flag names that entirely consisted of number. For example, "22" was (and still is) a legal flag name. This can create confusion when using some trigger methods: DML can't tell the difference between a number and a flag whose name happens to be a number.

To allow DML to distinguish between a number and **flags 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 the input to trigger only if the connected flag was equal to flag named 22, that condition would be

`="22"`

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

5.1.5.5 Bang! Output Methods
Since modules support strong (optional) methods for processing input signals, DML



also provides comprehensive methods to produce signals on outputs that mirror input conditions.

Since outputs in DML end on an exclamation point "!", I took a page out of shell scripting Lingo (see 'Shebang") and also call sending an output signal 'banging' a flag (it's the way my brain is wired; I'm aware that the verb "banging" makes some people uncomfortable. I think that those people need to grow up even more urgently than I). Like input trigger methods, mission designers can opt to choose from many different output (or bang!) 'methods'.

DML understands many methods that it can apply to a flag, and they mirror the input watchflag conditions we just discussed. Each method is defined by a keyword and/or an expression. When the module decides to bang! on the output, it applies the rules set forth below:

- 'on' **[set to 1]**
  Sets the flag's value to one, no matter what it was before. Same as using the number 1 (one)

- 'off' **[set to zero]**
  Sets the flag's value to 0 (zero), no matter what the value was before. Same as using the number 0 (zero)

- 'inc' **[increase by 1]**
  Increases the flag's value by 1 (one). Same as '+1'. If, for example, the flag's value was previously 10, that is increased to 11. **This is the most common bang! method and is the default for most DML module outputs**; it co-operates best with input flags that are set to trigger on "change" – which is also the default in DML

- 'dec' **[decrease by 1]**
  Decreases the flag's value by 1 (one). Same as '-1'. If, for example, the flag's value was previously 10, this number is decreased to 9.

- 'flip' **[alternate between 0 and 1]**
  This is a very effective methods 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. Note that this method can be error prone if one or more modules flip a flag's value multiple times before a module's input can detect a change: if two modules "flip" a flag in rapid succession before an input checks the flag's value, it appears to be the same as before even though it was changed: the second flip returned the flag to its initial value, masking the change. That is why the "inc" method is to be preferred over 'flip' for general flag bang!ing.

- #(number or flagName) **[set to absolute value or that of another flag]**
  Sets the flag to the fixed value (when giving a number) or copies the value from another flag, no parentheses.

  Examples:
  - #33 sets the flag to the number 33,
  - #-6 sets the flag to the number -6 (negative six),
  - #kills sets the flag to the value that the flag named 'kills' currently holds,

- `#"122"` (note the quotes) sets the flag to the value that the flag named '122' (a legal, old DCS flag name) currently holds

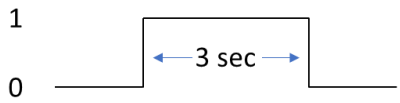
- `+`(number or flagName) **[add amount or another flag's value**]
  Adds the number give (no parentheses!) or the current value of the flag flagName to the flag.

  Examples:
  - "`+3`" adds 3 to the current flag value, while
  - `"+killScore`" adds the current value of flag "killScore" to the flag, and
  - `+"22"` adds the current value of DCS flag named 22 to the current value

- `-`(number or flagName) **[subtract amount or another flag's value]**
  Subtracts the number given (no parentheses!) the or current value of the flag flagName to the flag.

  Examples:
  - `-3` subtracts 3 from the current flag value, while
  - `-penalty` subtracts the current value of flag "penalty" from, and
  - `-"22"` subtracts the current value of numbered flag "22" from the current value

- `'pulse'` or `'pulse, <number>'` **[set to 1 and reset automatically]**
  "pulses" the flag by setting its value to 1 (one) for some time, and then re-setting it to 0 (zero) some time later. If you do not specify any time, the flag is reset after three (3) seconds. You can supply your own pulse time by adding a comma and a number,



  Example:
  - `"pulse, 4"` will keep the pulse up for four (4) seconds before dropping back to zero.

5.1.6 Further thoughts

When DML sends signals from an "output!" to an "input?", it uses *named flags* to carry that information. To connect an input to an output, you simply enter that flag's name for both the output attribute (the one whose name ends in an exclamation point "!") and the corresponding input attribute (the one that ends in a question mark '?'). That establishes the connection between output and input using the same named flag.

DML doesn't stop there. By default, inputs and outputs are configured to 'just work' when you connect them with a flag of the same name. For advanced mission designs, you can also control

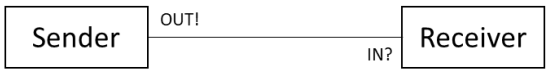
- The way a module sends signals. The **output method** describes *what is sent* (the value that is written to the flag that connects to an attribute that ends on an exclamation point "!") when the module wants to convey information. By default, this method is 'inc' (**inc**reases the flag's current value). You can opt to change that to other methods like 'On', 'Off', 'flip', a fixed value etc. A module sets an output's flag

whenever it deems is appropriate, usually as a response to something happening in the mission (a group enters a zone, an object is destroyed etc.)

- The way a module interprets their **input** (attributes that end on a question mark "?" in their name) and decides *when* it should trigger. Most common is the 'change' method: the module triggers when the flag's current is different from the last time it checked.

  Note:
  Modules periodically look at their input flags, usually once per second. This means that they don't immediately detect a signal/change; they *only see the change the next time that they look at their input flags*. This means that inputs can miss a signal if the input flag changes too quickly and reverts to the original value before the input had a chance to look at the flag. That is why DML prefers the 'inc' / 'change' combo for output and inputs: multiple changes can't accidentally revert the original value, the change will still be detected.

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.



OUT Methods
- Inc
- On
- Off
- Value
- …

IN Methods
- Change
- On
- Off
- Comparison
- …

- 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 results in *Receiver* to detect the change next time it checks, and then to trigger.

  Note:
  Any change is detected by Receiver – even if that change was not initiated by Sender. This is important to remember because advanced designs often connect multiple Senders on the same 'line' – like multiple alarm sensors can trigger the same central bell.

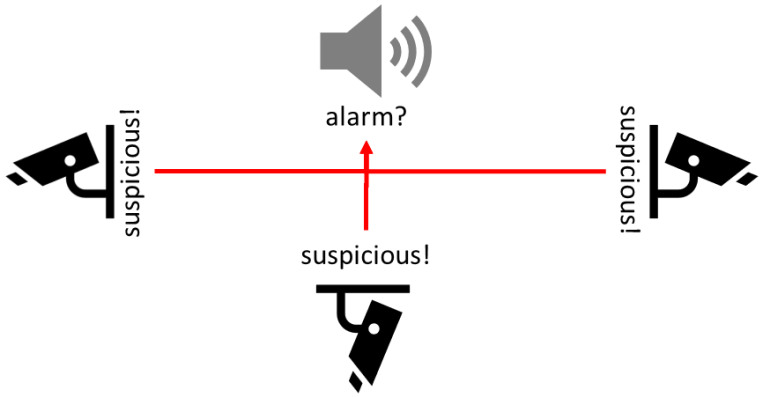
- Now let's change Receiver's Watchflag method to '>4', and assume that at the start of the mission, the underlying flag's value is zero (0). *Sender's* method stays at 'inc' (increment by one). The first four times that *Sender* sends a signal, the flag's value changes from 0 to 1, 2, 3 and then 4. Each time that the flag's value changes, *Receiver* sees the change when it checks, but ignores it because the condition ">4" of the flag's value is not fulfilled. The fifth time, however, *Sender* has increased the flag's value to 5. Upon inspection, *Receiver* detects the change of the flag's value, sees that the new value is ">4" and triggers.

Let’s now assume that Sender doesn’t increase that flag’s value for some time. When Receiver checks its input flags regularly, it will **not trigger when it checks the flag (usually once a second) because** although the flag's value is >4**, the value did not change** from the previous check. **Input flags require a change** on the watched flag in order to trigger. No, some time later, *Sender* does again increase the flag's value, now 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 that *Sender* triggers, it sets the output flag to On (1). *Receiver* detects the change, sees the new value '1', and triggers. The next time that *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.

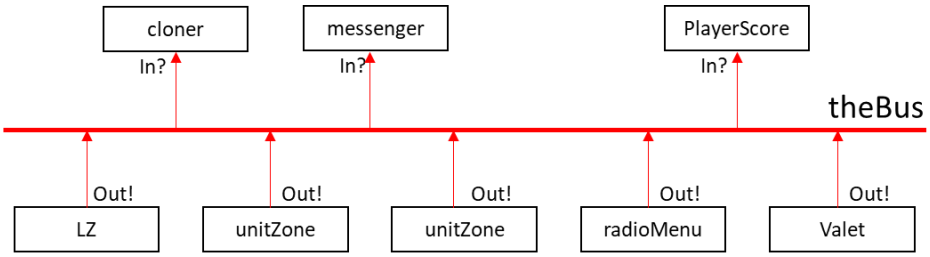
So here's the big take away: **Input flags are inspected periodically, and they require at least a change in value**. If there is no change detected in the flag’s value, a Watchflag does not trigger.

Now 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 any listening module that is waiting for 'change'. After that, the flag stays on the 'On' value (1). When the other modules trigger, they also set that flag to 'On'. To the listening receivers, though, the value of the flag stays the same, and no change is detected; nothing is triggered.
This is intentional and an immensely helpful design pattern to set up common scenarios: Like a ‘Burglar Alarm’ you can trigger a central reaction (alarm?) from multiple possible sensors that generate a signal (suspicious!) using the same ‘phone line’. The first sensor that detects something triggers a reaction, after that all others no longer elicit a response even if they send a signal; the alarm has gone off already.



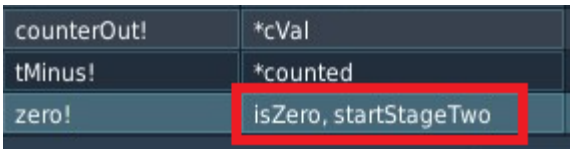
There is another important way to look at the way that we can ‘wire up’ our missions with DML, and it’s counter-intuitive at first: You don’t connect modules to each other; you connect listeners to a phone line and you connect speakers to a phone line. If a speaker and a listener happen to be on the same line, the speaker can talk to the listener. This is to say that you do not really connect one module’s input to another module’s output – that can be a coincidental result. You connect inputs to a line that can transport signals to the input. And

you connect outputs to a line that can propagate any signals that the output may generate. The important point: no module, knows nor cares about the existence of the other modules that are connected to the line. If the line is the same for inputs and outputs, then all inputs receive all signals that any output may put on that line. In engineer speak, this arrangement is often called a “Bus”.



So you can (and often do) have multiple modules connect their “Out!” ports to same named flag (called “theBus”) in above illustration). And you can have (and often do) multiple modules connect their “In?” ports to that same flag. All senders and receivers work independently from each other, some can connect or disconnect 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.

5.1.7 Multiple Output Flags



DML can bang! multiple flags at the same time. Unless otherwise specified, all outputs (those attributes that end on an exclamation point “!”) support this ability. To bang! multiple flags, simply list them as the attribute’s value and separate them by comma; leading/trailing blanks are ignored.

Please note the following:

- All flags are banged! with the same method.
- All flags are set at the same time, meaning that there is no guaranteed order in which they are changed.
- Should you list the same flag multiple times, it will get set multiple times; this means that the value changes internally multiple times; that does does not mean that inputs who watch that flag can detect multiple changes; from an external point of view the value of that flag changes at maximum once between inspections.

5.1.8 Attribute Synonyms – Why?

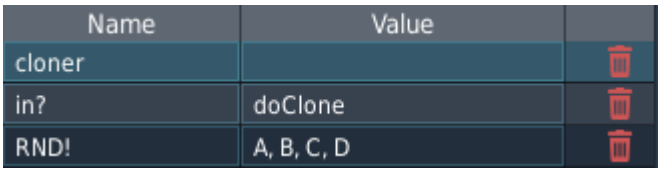
NOTE:
Attribute synonyms are under review and may be deprecated

Some modules offer multiple different names (‘Synonyms’) for the same attribute. For example, a clone zone can use the attribute ‘*spawn?*’, ‘*in?*’ and ‘*f?*’ all to trigger a new spawn

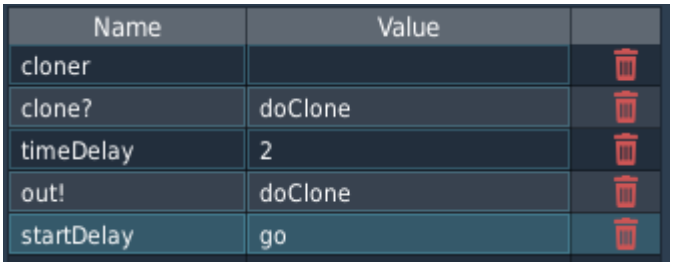
cycle. When you set up your spawn zone, you can use one, *and only one* of these per zone to trigger a cloning cycle.

Cool - but *why*?

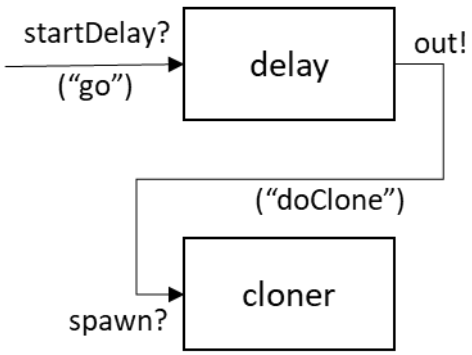
The reason for this is to facilitate module stacking. In DML 'module syntax', many modules support a generic '*f?*' or '*in?*' as input. If you stack modules on the same zone, you often want them to trigger at the same time. Use a single '*f?*', and all anchored modules that understand '*f?*' as input 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 random generator. Both cloner and RND support '*in?*' as input (here set to flag named "doClone"). When flag doClone changes, both the cloner and the messenger modules activate and do their thing.



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 'go' by 2 seconds before passing it to '*out!*' on flag 'doClone'. Since 'doClone' is 'wired' into the cloner's '*clone?*' input, this causes the cloner to start a clone cycle 2 seconds after the timeDelay received a signal.



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.

5.1.9 DML Flags: Named, and Local Flags

One of DCS ME's less loved aspects was the fact that historically, Mission Editor flags only used numbers as their names. This has changed with a late version of DCS 2.7 – now DCS ME also supports named flags, but many existing missions still use numbers to name their flags: be it because the mission never was updated or because the mission designer prefers old-school flag naming.

If you use numbered flags, it is up to you to remember that flag “37” signifies that the enemy Flanker has been triggered, while “45” signals that the tanker is on-station. Needless to say, I prefer named flags. If you like numbered flags, when you mix them with DLM I recommend that, just to make clear that you are referring to a flag precede the number with the letter “f”, e.g. instead of naming a flag “123”, name it “f123”. That will help you to distinguish the name from a value – something that many not currently be important in plain vanilla ME, but will make things much cleaner (and clearer) should you start using DML methods.

5.1.9.1 NAMED FLAGS



DML supports flags with any name. There are a few rules to adhere to (→ Flag Naming Rules), but in DML you can give meaning to flag names. Although discouraged, DML is backwards-compatible to old-style number-only *DCS* flags: you can use a flag name that entirely consists of digits (e.g. “123”), as it was used in older versions of DCS. DML’s flags are fully compatible with DCS/ME’s 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 alphanumerically named flags.

A WORD OF CAUTION – TO OLD-SCHOOL DESIGNERS
With expanding DML abilities, old-school number-only flag names are still supported, but discouraged (if not positively frowned upon). One reason is that -unless you are very careful – when using some of the more advanced Trigger Methods, you *can easily* mistake a number-only flag for a number. DML provides “quoted” number-flags as a work-around, but this is error prone for most mission designers.

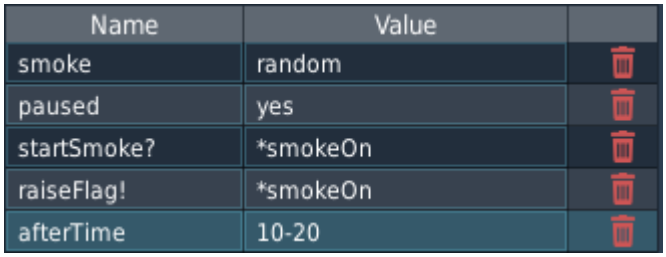
I **strongly recommend** that you - whenever possible - change old-school number-only flag names to flag names with a leading “f”: “123” thus becomes “f123”. This avoids confusion with numbers, points to the fact that the name has legacy background, and is fully compatible with both DML and modern DCS.

Note that some DML modules support a QoL feature called “flag ranges” (e.g. “10-20”) which *only* works on number-named flags. If you intend to use flag ranges, you should weigh their naming convenience against possible confusion. Numerical flag ranges are cool, yes. But are they worth the risk?

*5.1.9.2 ZONE-LOCAL FLAGS: *name*

DML provides mission designers with a powerful feature that elevates flag use above what DCS provides to the masses: **local flags**. Every mission designer who has spent just a little time designing DML-enhanced missions has also assembled some small ‘Zone Automatons: a small stack of modules that talk to each other *inside the same zone*. This often involves a

mix of delay, counter, messenger, logic gate or randomizer modules. One challenge with these “automatons” is that they require one or more unique flags to communicate with each other, and can make them ungainly to work with, **especially when using copy/paste.**



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

Zone-local flags are *only visible to modules that anchor to the same zone*. They are invisible to modules in other zones (and hence can use a flag with the same without conflict), DCS/ME or Lua.

And why Is that important? Copy&Paste. If you look at the example of the delay-started smoke zone above, you’ll notice that you can copy and paste the entire trigger zone all over the map, and don’t have to change a single attribute. Wherever you place such an ‘automaton’, there will be a randomly-colored smoke marker starting up 10 to 20 seconds after the mission starts.

5.1.9.3 FLAG NAMING RULES

You have great freedom in naming your flags, and may even use old-school ‘number-only’ flag names. To work well with DML, here are some flag-naming rules that, should you adhere to them, guarantees that they integrate well:

	DML (and ME)	**DML Zone-Local**
Format	• Alphanumeric • must not contain comma ‘,’ • must not start with asterisk ‘*’ • must not start with double quote ‘”’. • *should* not start with a digit (‘0’...’9’)	**Starts with asterisk** ‘*’, alphanumeric, must not contain comma ‘,’
Examples	• A12 • With blank • F***d up • Yup “quotes” too	• *1 • *A12 • *fireCloner • *ok multi **
Scope / Visibility	DML modules, Entire DCS (newer versions)	Only DML modules attached to the same zone
Invisible to	n/a	Everyone outside Zone

5.1.9.4 DML FLAG RANGES

There is one case in which “classic” ME number-only flags make sense, and I recommend that you make use of it when is suits your needs: for 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.

5.2 Special Concepts

There are a couple of DML-specific concepts that only apply to some modules, and that you can skip until you need them. Most of these revolve around spawners (not cloners): typeStrings and Orders.

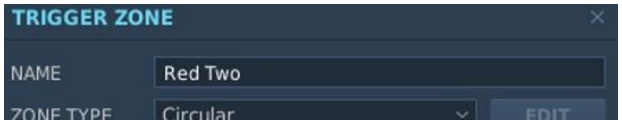
5.2.1 Orders (Spawn Zones)

Ordering troops is an ability several modules support in DML: modules that can produce (spawn) units, and that are able to give, or pass on, “DML 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.

5.2.1.1 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’. Units that are under “attackOwnedZones” orders will strive to use roads, and therefore ignore a *moveFormation* attribute.	
captureAndHold	Like “attackOwnedZone”, except that the units choose the nearest enemy/neutral zone once, and then move to capture it. If the zone is captured, the units continue to move to the zone,	

Orders	Description	Parameters
	and then remain there, switching to “guard” orders. Respects *moveFormation*	
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
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	

5.2.1.2 “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.

5.2.2 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 that 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)
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

5.2.3 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.

5.2.4 Zone Ownership / Owned Zones

Zones ownership is a DML concept that associates zones with an 'owning' faction (Note: this is a DML feature and is currently not available in ME). DML modules can use this ownership information to modify their behavior, e.g.:

- Change in ownership can trigger flag changes
- Spawners can stop spawning when their zone controlled by the wrong faction
- Cloners can inherit the owning coalition and apply ownership to the cloned units
- Troops can be ordered to seek out and attack the nearest enemy owned zone
- FARP Zones automatically handle ownership change and spawn resources
- Owned zones can inherit ownership from other zones.

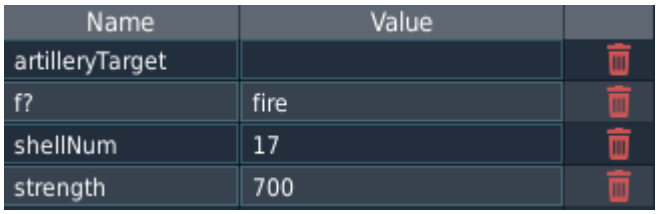
Unless you specifically add modules that manage a zone's ownership (e.g. "OwnedZones"), a zone's owner remains static throughout the mission.

When using the “Owned Zones” (see that section in this documentation) module in your missions, a zone’s ownership can become dynamic and change during the mission: they can be 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.

5.3 “Building Blocks” Modules

All DML modules provide key functionality which 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 to offer several important features

- Use ME’s GUI to place a module’s functionality, including copy/paste to rapidly populate a map with functionality.
- 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 change a module’s global configuration/setup.
- 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, its 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 modules 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)

5.3.1 Core Abilities (shared by all DML Modules)

Since all modules internally build on DML's foundation modules "dcsCommon" and "cfxZones", all modules inherit some core abilities. These are:

5.3.1.1 Location and Size

All DML zones use the Mission Editor to place and move zones. The location of the zone (which can be used by many zone enhancements to place their effect) is usually it's center. For example, Smoke Zone places the smoke effect at the center of the Zone. Some DML Zone enhancements can use the Zone's surface ("size", circle or polygon) for its purpose (e.g., Owned Zones, Unit Zones), while others ignore the zone's size entirely (a smoke zone does not care how big the zone is, the smoke effect is always dead center, and the same size).

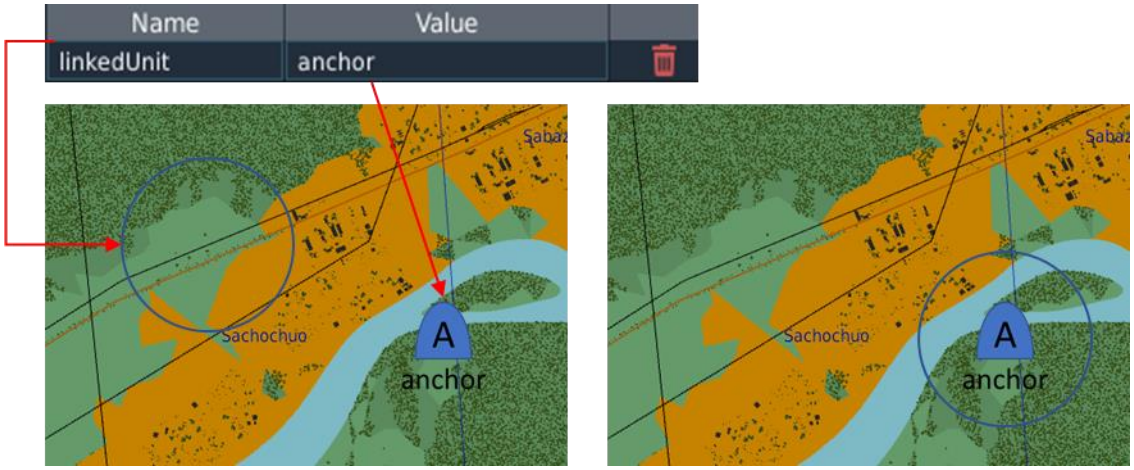
5.3.1.2 Moving Zones (linking zones to units)

DML significantly expands on DCS' "moving zones" concept. DCS allows a 'moving zone' that is attached to a unit in such a way that the moving zone always moves with the unit, centered on a unit.

DML takes a more refined approach. Just like with DCS, in DML any zone can become a moving zone, and most modules support a dynamic location (to the extent where it makes sense and is not prohibited by the underlying DCS core)

Centered on Unit (mimics classic DCS)

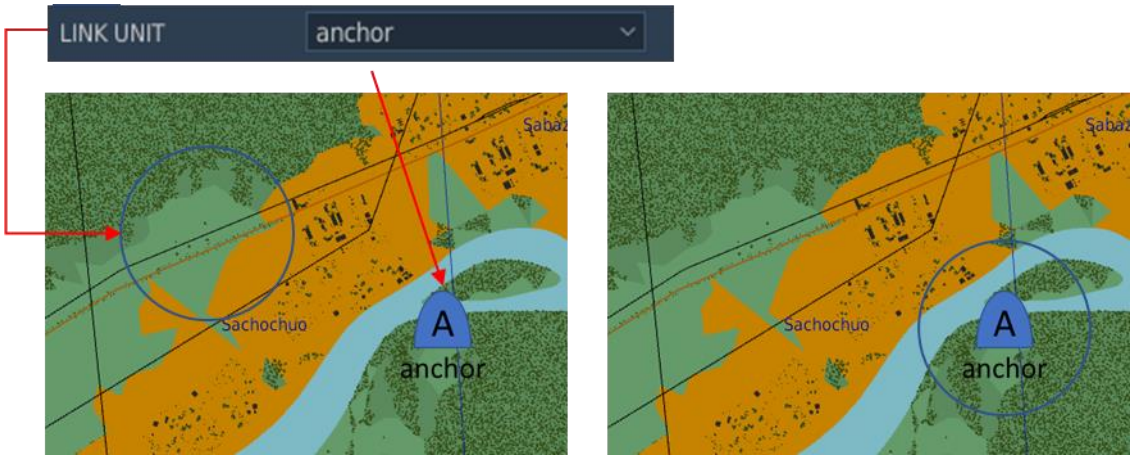
To make a zone follow unit, either add the "**linkedUnit**" attribute with the value containing the name of the unit that this zone is linked (bound) to, or use the Zone's "LINK UNIT" pop-up (above the attributes block) and choose the unit to link to. When you are using the linkedUnit attribute, the name of the unit **must match exactly** (when you use the LINK UNIT pop-up Mission Editor takes care of this for you). In the example below, the zone is linked via the *linkedUnit* attribute to a unit called "anchor" (note the lower-case 'a') – DCS is case sensitive, and this extends to matching unit names.



If there is no unit “anchor” in the mission at that point in time, no link is established, and the zone remains where it is. Additionally, if you have enabled that zone’s verbosity, DML outputs a warning message

```
Linked unit: no unit <anchor> to link <Intercept> to
```

In the example below, we now link the zone via Mission Editor’s zone LINK UNIT pop-up to the ‘anchor’ unit:



Again, if the linked unit doesn’t (yet) exist, the zone remains where it is until a link can be established.

Note that you can use either method (attribute or pop-up) to link a zone to a unit, but not both methods for the same zone. If you do that, DML warns you when the mission starts:

```
WARNING: Zone <dual> has dual unit link definition. Will use link to unit <T1>
```

When the mission starts, the zone is moved to center on the linked unit and moves with it. If the linked unit does not exist (be it because of misspelling, or the unit does not *yet* exist), a moving zone will not move (and remains at its ME-placed position). If a moving zone’s linked unit disappears, the moving zone stops moving and remains at the last location it was moved to (i.e., it does NOT revert to its initial position). If, at a later point a new unit is created with the same name (even though unit names are unique in DCS, and there can only be one unit with any given name at the same time, this is a common occurrence with “client” or player units that can disappear and re-appear at any time during the mission), the zone ‘snaps’ to that unit.

If DML can link a zone to a unit, and you have that zone’s verbosity enabled, DML outputs a notification when it links the zone to a unit (as mentioned before, this can happen multiple times during the same mission):

```
Link established for zone <Intercept> to unit <anchor>: dx=<2635.2774043254>, dz=<-1858.
0660008399>
```

If you link a zone to a unit without adding any other attributes (see below for more), the moving zone behaves like DCS: it’s always centered on the linked unit.

DML adds several powerful capabilities to moving zones that you enable by adding attributes to the zone that you link to a unit:

When to use 'linkedUnit' attribute versus "LINK UNIT" pop-up
There are advantages and disadvantages to both methods, so choose either depending on your needs:

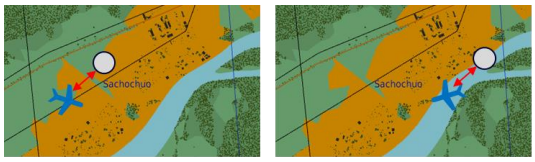


The advantage of using a *linkedUnit* attribute over ME's pop-up method is that the linkedUnit attribute can be used with *dynamically spawned units* (i.e. units that aren't placed with ME). The main disadvantage when using the attribute is that it can be prone to spelling mistakes, and when you change the name of the linked unit's name in ME: the attribute isn't automatically updated, and the link can become broken.



The advantage of using the LINK UNIT pop-up is that it's easier to spot visually in ME (it's placed prominently above the attributes box), more user-friendly (the pop-up menu gives convenient access to all units that you have placed in ME), and that it automatically updates a linked unit's name should you change the unit's name in ME. It's main disadvantage is that it can only link to units that you have already placed in ME, so it is impossible to link to units this way that spawn dynamically.

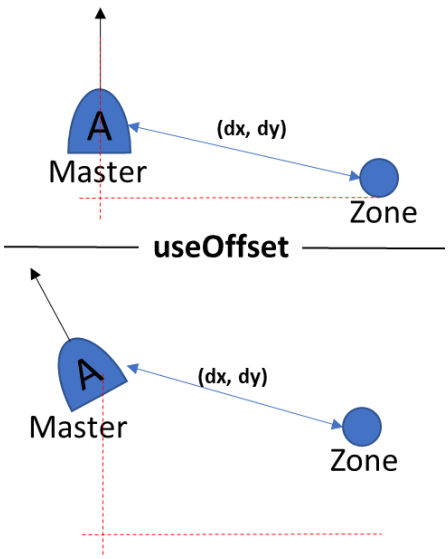
Have zone move with a unit while keeping offset
Instead of always centering on the linked unit, you can move the zone relative to the linked unit.



Zone always stays at the same map offset to master, irrespective of that unit's heading

To turn on this ability, add the "**useOffset**" attribute with value 'true' (note: you must also have *linkedUnit* set). Such a zone will always remain at the same relative offset (in N/S and E/W direction) to the master unit (*linkedUnit*). The offset is taken from the map when the mission starts. So, if you placed the master unit 100 meters to the North and 300 meters to the east of the zone, DML makes sure that it remains in that relative position. If the unit moves 2 miles to the Northeast, so will the zone.

The zone therefore always remains in the same location relative to the linked unit *as seen from the map, independent of the unit's heading*.



Zone "Orbiting" the linked ("master") unit

Often you want a zone to move at the same distance from a unit and want the zone to remain at the same *relative bearing* to the unit. In other words: if the zone is to the right of the master unit as seen from the way it is heading, it should remain on its right wherever it is turning.



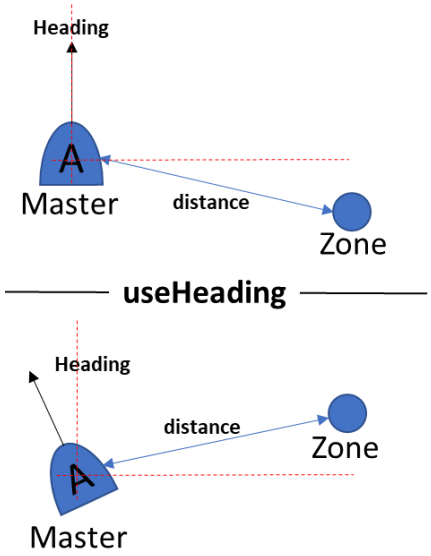
Zone initially set at 4 o'clock position rotates with master unit to stay at 4 o'clock

Put differently: the zone orbits around its master in order to keep its position. The zone moves with the linked unit, and it turns with it, with the unit at the center of the circle.

DML supports this when you add the "**useHeading**" attribute with value true. Now, the zone always remains at the same bearing from the master unit, moving with the unit and (more importantly) turning with the master unit.

In the example to the right, the top depicts a zone and its master at their initial position: the Zone is (distance) units away from the master unit and sits at the unit's 4 o'clock position. In the lower image, the master unit has turned some 45 degrees to the left. To remain at the 4 o'clock position, *the zone rotates (orbits) around the master unit for the same amount (45 degrees).*

Note that **the attributes *useHeading* and *useOffset* are mutually exclusive**; use only either. Should you use both, the results are unpredictable.



Moving Zones with Player/Client (and late activated) Units

If you link a zone to a player/client unit you should remember that *at the start of a mission* (when cfxZones initializes and attempts to move all moving zones to their destination), *player and client units do not yet exist*. This is **not** an impediment to zones that are linked to a unit, as DML supports late linking to units. The consequence, however, is that the zone is not moved to its intended location until the relevant unit enters the game (which may be much later in the mission, or not at all).

A side effect is that when you link a zone to a player unit, and that unit is first destroyed and the player then later re-slots to that unit, the linked zone remains at the last known location (the location where the player unit was destroyed) until the moment when the player re-slots into that unit – at which point the zone teleports to the player's 'replacement' unit's location.

5.3.1.3 Ownership (Coalition)

All zones carry an implicit 'owner' attribute that is used with many modules to establish who owns the zone. You can force the value of this by setting the 'owner' attribute in Mission

Editor, but unless you are using the OwnedZones module or some other that explicitly requests that you set it, you shouldn't set a zone's owner attribute in ME.

5.3.1.4 ME Attributes

Name	Description
linkedUnit	Moves the zone's center with the unit whose name *exactly* matches the value of this attribute. That unit must exist at the beginning of the mission, or the linked zone remains at its ME location until the unit exists and becomes linked (at which point it moves to the unit's location). If the *linkedUnit* ceases to exist after the zone has followed it, the zone remains at the last location it moved to until a new unit with that exact name appears again. This is often the case with player-controlled planes, so expect this to happen and design your mission accordingly. If neither *useOffset* nor *useHeading* attributes (see below) are set to true, the zone always centers on *linkedUnit's* location. **Note:** be advised that all player and client units do not exist at the beginning of a mission and spawn only when a player occupies that slot. This means that zones linked to player or client unit will only move to those locations after the player has entered the game. **Note:** The 'linkedUnit' attribute is one of two methods to link a zone to a unit. The other method is ME's dedicated 'LINK UNIT' pop-up (see description, above). Use only one of the two methods to link a zone to a unit, else DML will give you an error. The advantage of using a *linkedUnit* attribute over ME's pop-up method is that the *linkedUnit* attribute can be used with dynamically spawned units (i.e. units that aren't placed with ME). Its main disadvantage is that it can be prone to spelling mistakes or when you change the name of the linked unit in ME. Defaults to <not linked to any unit>
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 linked 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. **Requires either *linkedUnit* or LINK UNIT be set** Not compatible with *useHeading* (below) – use either Defaults to <not set>
useHeading	When set to true, the zone moves and turns in synch with and relative to the *linkedUnit*. **Requires either *linkedUnit* on LINK UNIT be set**

Name	Description
	Not compatible with *useOffset* (above) – use either Defaults to <not set>
owner	The coalition that owns this zone. Used with some zone enhancements. Do not set this attribute unless you know what you are doing, or a module's documentation requests you to do so. Defaults to neutral

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

5.3.2 smokeZones

DCS already provides colored ‘smoke’ effects (various colors) that unfortunately end (dissipate) after some time (5 minutes). For many missions it may be useful to *permanently* mark a location with colored smoke.

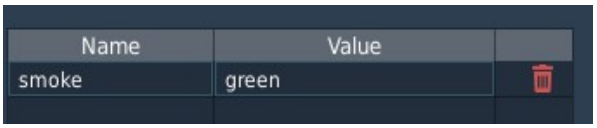
cfxSmokeZones does exactly that: place (permanent by default) smoke (at the trigger zone’s center) with the color that you set as the attribute’s value.

Note:
Since this is the very first module covered in the manual, I’m being a bit more verbose when I explain how to use ‘smokeZones’. Please bear with me.

5.3.2.1 Description

The zone’s center hosts a “permanent” **colored** smoke effect that either starts when the mission begins, or (should you tell it to wait), starts at the mission’s command..

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



You add zones to your mission by using Mission Editor (ME), and then (again using ME) you add the required attribute(s).

By default, a smoke zone starts emitting smoke from its center when the mission begins that (unlike DCS’s built-in ‘colored smoke’ effect) does not dissipate unless you tell it to.

You can tell a smokeZone to start and stop emitting smoke using commands/signals sent to it’s inputs. Note that although smoke zones respond immediately to a command received on “startSmoke?”, it can take up to 5 minutes for DCS to respond to a command received on “stopSmoke?”.

Note also that smokeZones emit colored smoke without flames. A sister module *fireFX* provides visual effects reminiscent of fires (flames, black smoke).

ME INTEGRATION

You can start the mission with the smoke turned off, and then turn on the smoke effect later. To do so, first add an attribute *paused* with value *true* (this causes the smoke to be turned off at mission start). Then add an attribute *startSmoke?* (remember the question mark “?”) and add the name of the command (e.g. *smokeOn*) that you want this smokeZones to listen for. When it hears this command, smoke starts emitting from the center of this zone.

There are additional attributes that you can use to further customize the smoke that emits from the zone. For example, the agl attribute controls how far above the ground smoke emits. You can use this for some special effects, e.g. colored smoke emitting from chimneys, or to mark landing zones on the roofs of high-rises etc.

MULTIPLAYER ODDITIES
Like the rest of DML, Smoke Zones fully support multiplayer. There is a known, longstanding DCS bug in multiplayer that existing smoke effects are not transmitted to clients that connect. So it your players connect to a mission that uses active smokeZones, they may take up to 5 minutes to be visible on clients. This bug does not extend to smoke that is started after clients connect – all clients can see newly started colored smoke effects immediately.

5.3.2.2 Dependencies
Requires dcsCommon, cfxZones

5.3.2.3 Module Configuration
None.

5.3.2.4 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) Defaults to "green" **MANDATORY**
paused	When true, this trigger zone will not start smoke at mission start, but wait for a signal on the *startSmoke?* flag. Defaults to false (smoke starts at mission beginning). Note that if you set *paused* to true and omit a *startSmoke?* attribute, you cannot start this zone's smoke in the mission.
startSmoke? ~~f?~~	DML input to listen for a command/signal. When this input hears/receives the specified command/signal, smoke starts to emit from the zone (if it wasn't already emitting before, no change otherwise). Defaults to <none>
stopSmoke?	DML input to listen for a command/signal. When this input hears/receives the specified command/signal, smoke is scheduled to stop emitting from this zone. Note that **it can take up to 5 minutes** for the colored smoke to stop (unlike *fireFX*, there are no provisions in DCS for colored

Name	Description
	smoke to be extinguished immediately, so we have to wait for it to time out). Defaults to <none>
agl alt altitude	Altitude (in meters) above ground that the smoke should be created. Defaults to 1 meter
triggerMethod smokeTriggerMethod	Conditions when the inputs should trigger. Used with the “Flag Model” Defaults to ‘change’

5.3.2.5 Using the module

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

Add a trigger zone to the mission and add the ‘smoke’ attribute to that zone, with a value for the smoke’s color.

5.3.3 rndFlags

5.3.3.1 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 to randomly set flags and make your missions more random.

A classic setup is a mission where you have different groups of enemies, and you want to randomly activate some of them – with “some” meaning an element of surprise. Doing this with triggers *is* possible, but near-nightmarish to maintain, and full-nightmarish to service later.

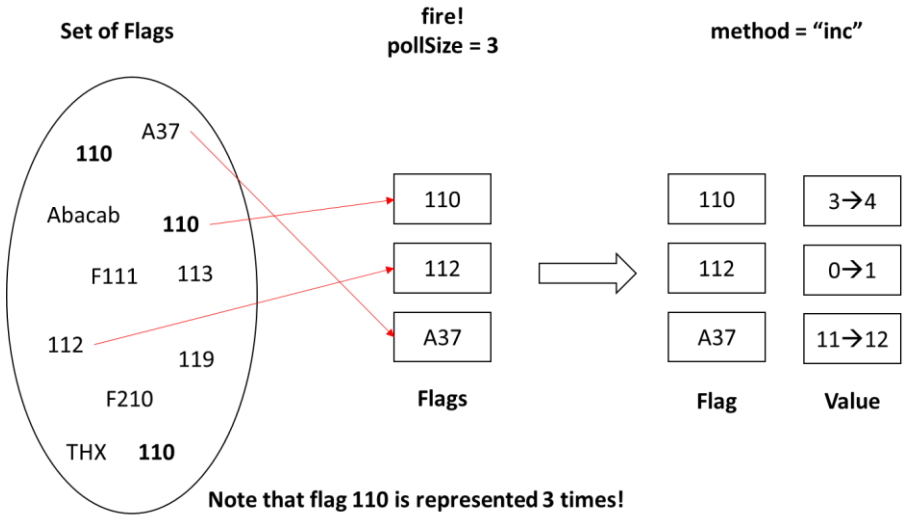
With rndFlags it is just a matter of placing a zone and adding a couple of attributes. rndFlags are exceedingly easy to use, and incredibly flexible.

Basic Function

Since randomizers are position-independent, place them anywhere on the map. I place them close to the things that I want to randomize, but you can place their trigger zones anywhere.

A rndFlag randomizer functions like this:

- 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 to cycle (by triggering its input, or at mission start), and the randomizer picks one or more flags from the set each time that the randomizer fires, and then bangs! on these flags according to the DML ‘method’ that you chose for this randomizer.



Starting from this simple mechanism, rndFlags can do a lot:

- The set of flags that the randomizer chooses from is flexible, and you can add the same flag multiple times so it gets chosen more often
- The randomizer can pick more than one flag at once
- After picking flags, you can let the randomizer 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. That way you can make sure that all flags eventually are polled, but in a randomized order.
- You have multiple options to tell the randomizer what to do with the flag it chooses (what method to bang! on the flag)
- You can trigger the randomizer as often as you like

Set of Flags
You start with a set of flags that you pass to the randomizer with the 'RND!' (main randomizer) attribute. As value to that attribute you list the flags to include as follows:

- Separated by comma (e.g. "3, G, SAMs, 5,")
- the order in which flags are listed is not relevant
- you can also include number ranges by giving the lower and upper bound separated by a hyphen (e.g. "10-20"). This is only applicable for old-style DCS (number-only) flags, and something you should avoid
- 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, SAMs, 4"). This will add that flag **multiple times** into the set, making it more likely that it is chosen.
- 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 "SAMs" to have a 50% chance to be chosen, and flags 6 and "B" each only a 25% Chance. You achieve that simply by creating a set "SAMs, SAMs, 6, B" for the randomizer

Poll Size (optional)
Each time the randomizer is triggered, it chooses a number of flags from the set of flags that you provided. 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.

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

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

Remove and Reshuffle (optional)
By default, each time the randomizer is triggered, the set of flags to choose from is the same as the set it starts with. You can change that by enabling the 'remove' option. If set to true, each time that the randomizer runs, it removes the item(s) that it chooses from the set. For example, if the start set is "5, SAMs, 9", and the randomizer picked the "SAMs" item, the next time that the randomizer is triggered, 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 checks if you also set the 'reshuffle' option to true. If reshuffle is true, the randomizer "reshuffles" all flags by refilling it the original full set of flags.

If reshuffle is disabled, the randomizer simply does nothing; all flags are exhausted, and there is nothing more to do.

Flag Change Method (optional)
After the randomizer has chosen the flag(s) to work with, it applies the DML method to them that you set up (by default that is the 'inc' method that increments a flag's value by one).

Triggering a randomizer cycle
There are multiple ways to have a randomizer run through 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.

- `rndPoll?`
  This is a DML watchflag. The randomizer observes this flag for value change, when the value of this flag changes in the way that matches the 'triggerMethod', the randomizer triggers (undergoes a randomizer cycle). By default, a randomizer's triggerMethod is 'change', so any change in the flag's value will trigger a cycle.

Note that if you choose **neither onStart nor rndPoll?** 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.

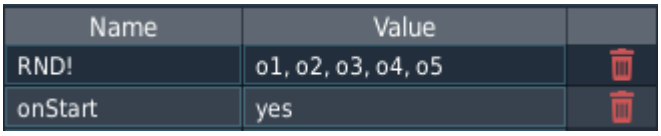


The randomizer on the right creates a set of five flags: "o1, o2, o3, o4, o5".

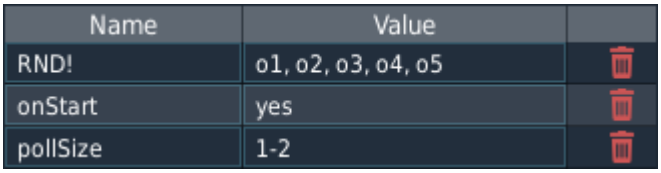
Since there are no other attributes, the following happens when the mission starts:

- No pollSize attribute is present, so pollSize is set to 1 (one)
- No “rndMethod” attribute is present, so the output method is set to “inc”, all flags in the set will be value-increased by one when they are chosen
- Neither “onStart” nor “f?” attributes are given, which causes onStart to be automatically set to true: the randomizer runs exactly once at mission start.
- Since onStart is now true, a cycle is scheduled for 0.25 seconds after mission start
- At 0.25 seconds mission time, the randomizer cycles
    - A flag from the set “o1, o2, o3, o4, o5” is chosen
    - The value of chosen flag is increased by 1 (method = “inc”)

While the above example nicely illustrates the amount of engineering and QoL features that have gone into designing it, a better, more explicit way of creating the same randomizer would be by adding the onStart = true attribute to show that it was your intention that the randomizer runs once on mission start.



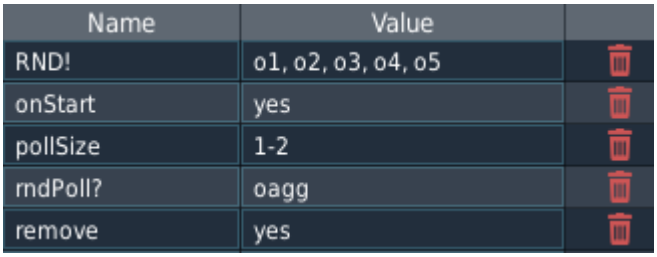
Now let’s get a bit more fancy:
we have a mission with 5 groups of enemies. Each time that the mission starts, we want that one or two of them activate. Each group is activated by their own flag, o1, o2, o3, o4, and 05.



We add one (two) additional attributes: pollSize (and onStart). Let’s see what happens when we run the mission:

- No “rndMethod” attribute is present, so the method is set to “inc”
- “onStart” is set to true (“yes” and “true” are the same for DML)
- Since onStart is true, a cycle Is scheduled for 0.25 seconds after mission start
- At 0.25 seconds mission time, the randomizer cycles
    - pollSize is set to a random number between 1 and 2, including 1 and 2. Let’s assume 2 is chosen
    - Two flags from the set “o1, o2, o3, o4, o5” are chosen
    - The two chosen flags’ values are each incremented by 1 (rndMethod = “inc”), which activates the groups that are connected to the two flags

Now let’s go wild. We have five enemy groups, each activated by their own flag o1, o2, o3, o4, o5.



We use the flag “oagg” (short for “opposing aggressors”) to trigger the randomizer, and each that time we run a cycle, we want to activate 1 or 2 of these groups. When the mission starts, the first batch of enemies should activate. Also, once a group has been activated by the randomizer, it should not be activated again.

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

- No "rndMethod" attribute is present, so the method is set to "inc"
- Since onStart is true, a cycle is scheduled for 0.25 seconds after mission start
- Since no "triggerMethod" is set, it defaults to "change", the randomizer will cycle each time the flag's value connected to the input changes
- The input "rndPoll" is connected to the flag named "oagg". The randomizer is now primed to trigger a cycle each time that the value of the flag "oagg" changes
- At 0.25 seconds mission time, the randomizer cycles (due to onStart = true)
    - pollSize is randomized to a value between 1 and 2. Let us assume that "1" is chosen
    - One flag from the set "o1, o2, o3, o4, o5" is chosen. Let us assume it is "03"
    - The chosen flag o3's value is incremented by 1 (rndMethod = "inc") - which activates the group that is controlled with this flag (in DML, you'd often connect a DML spawner or cloner to do that for you, see those chapter if you are intrigued)
    - The item "o3" is removed from the set for this randomizer, the remaining set is "o1, o2, o4, o5"
- (some time later) flag oagg changes its value, causing this randomizer trigger:
    - pollSize is randomized to a value between 1 and 2. Let us assume that "2" is chosen
    - Two flags from the (reduced) set "o1, o2, o4, o5" are chosen. Let us assume they are "o1, o5"
    - The values of chosen flags "o1" and "o5" are increased by 1 (rndMethod = "inc"), which activates these groups in ME (or via DML cloners, spawners, …)
    - The items "o1" and "05" are removed from the set for this randomizer, leaving "o2, o4" as remaining set
- (this repeats until all items are removed from the set)
- (a little later yet again) flag oagg changes its value, causing this randomizer to cycle
    - The randomizer notices it has no items left in its set and does nothing.

Further Considerations

rndFlags allows you to quickly add the ability to change several randomly selected flags from a set of flags. If you were to build this in plain-vanilla Mission Editor, this requires that you first add triggers to your mission that watch these flags and assemble conditions to match the desired trigger situation, and then perform actions accordingly (like, for example, activate a group) when a flag changes.

Many DML modules (those who support input attributes (that end on a question mark '?'), for example cloners and spawners, watch flags autonomously and initiate their specialized action without you having to set up ME triggers. You can quickly create a highly randomized mission by placing a bunch of Clone Zones on the map, and 'wire' their input flags to the output flags controlled by a randomizer.

One of the most powerful mission design concepts is to combine couple of randomizers (rndFlags module) with a pulser (pulseFlags) and Clone Zones (cloneZones) to quickly create an unpredictable, near-infinite range of mission set-ups.

ME INTEGRATION
This module is all about ME integration. Use it to be able to feed flag names into the module, and then trigger these flags as described. You can place the zones wherever you please, as their function is not location-dependent. I recommend that you place them in proximity to the modules that you want to control with the randomizer. To control a randomizer's operation connect a flag to its input "rndPoll?"

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

5.3.3.2 Dependencies
rndFlags requires dcsCommon and cfxZones

5.3.3.3 Module Configuration
To configure rndFlags via a configuration zone,

- Place a Trigger Zone on the map
- 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

5.3.3.4 ME Attributes

Name	Description
RND!	Marks this as a randomizer. Value describes a set of flag names, separatzed by comma (',') that the randomizer chooses from. The flag names can appear in any sequence. Supports numerical ranges like "2-7" (for old-style ranges of number-named flags only). Flag names can be included multiple times; including the same flag name multiple times simply increases the likelihood that this name is chosen. Examples: "2, 4, A, A, F, 6" "A9, 3-18, C33, samAttack, 11-11" **MANDATORY**
method	DML Output flag method.

Name	Description
rndMethod	Defaults to “inc”
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 input Watchflag. When triggered, starts a random cycle. Defaults to <none set> You can use any synonym, but only one per Zone
triggerMethod rndTriggerMethod	DML Watchflag condition when to trigger inputs. 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 rndPoll? (nor synonym) is specified, and onStart is false, the randomizer will never activate. You’ll receive a warning. **NOTE:** If you specify neither onStart nor an input rndPoll? (or synonym), onStart is automatically set to true, so that the randomizer runs exactly once, 0.25 seconds after mission start.
done!	DML output. The flag names to bang! when the randomizer has run out of flags to change, and reshuffle is false (randomizer did nothing) Is banged! every time that the randomizer runs a cycle on an empty flag set

5.3.3.5 Persistence

RNDFlags persist the current flag pool if you have chosen to remove flags each time you run a random cycle. Do if you started with (a, b, c, d) and already ‘spent’ the ‘c’ flag, upon loading the randomizer only has the flags (a, b, d) at its disposal. Reshuffle will reset to full original stock.

5.3.3.6 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.

5.3.4 pulseFlags

5.3.4.1 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). You can randomize the number of pulses generated by supplying it with a range (e.g. 3-7), and the pulser will generate a random number of pulses within that range.

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 'fire' once every two seconds.



When run, the pulser on the right works as follows:

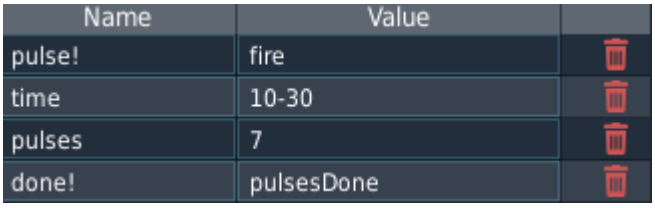
- When the mission starts, it checks if it should send out a signal before it starts the timer for the first pulse. If zeroPulse is true (which it is by default, and since we don't change it, we want a zero pulse), it sends out a signal to all flags that are connected to the pulse! output. In our case it is the flag named 'fire', and so the value of that flag (which is zero at mission start) is incremented by one (1).
- 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 randomly 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, let's create a pulser that changes flag 'fire' exactly 7 times, with 10 to 30 seconds between each pulse, and then increment flag 'pulsesDone' when it is done



Starting a pulse
When the mission starts, a pulseFlags zone automatically starts pulsing if it does not have its "*onStart*" attribute explicitly set to *false* (meaning: by default a pulser starts to pulse on mission start, and to prevent it from doing so, you add the *onStart=false* attribute). A paused pulser can be started with a signal sent via flag to the *activate?* input. It remains dormant until it receives a signal on that input.

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

Name	Value	Description
pulse!	Name	This are the flags that gets changed according to method each time that the pulser cycles.
done! pulsesDone!	Name	Output flags to change when the number of pulses as defined with the 'pulses' attribute have completed.
activate? startPulse?	Name	Watchflag that (re-) starts a paused pulser (with the first pulse)
pause? pausePulse?	Name	Watchflag that when triggered, stops the pulser and puts it into paused state

5.3.4.2 Dependencies
pluseFlags requires dcsCommon and cfxZones

5.3.4.3 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

5.3.4.4 ME Attributes

Name	Description
pulse!	Marks this as a pulser. The value connects the flags to change on each pulse to this output. The flags are changed according to the method attribute. **MANDATORY**
method pulseMethod outputMethod	DML output method. Use only one synonym per zone. Defaults to “inc”
done! pulsesDone!	DML Output: flags to change (according to DML output method) when the pulser has run through the entire number of pulses. Can only happen when the pulses attribute supplies a positive number. Defaults to <none>
activate? startPulse?	DML input Watchflag. When triggered, a paused pulser is reset and then restarted. Use only one synonym per zone. Defaults to <none>
pause? pausePulse?	DML input Watchflag. When triggered, a pulser is paused. Use only one synonym per zone. Defaults to <none>
onStart	When set to false, a pulser starts paused, else active. **Defaults to true** (pulser starts automatically)
pulses	The number of pulses to complete. • If set to -1, the pulser runs until the mission ends or the pause?-flag changes • If set to a number, the pulser will generate that many pulses. • If set to a range (e.g. “3-5”) the pulser will generate a random number of pulses within that range. Defaults to -1 (endless)
time pulseInterval	Number of 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 (second)
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 sent out immediately)

5.3.4.5 Persistence

PulseFlag automatically supports persistence. It restores the number of pulses left and re-initializes a running pulse to the number of seconds left.

5.3.4.6 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.

5.3.5 delayFlags (“Timer”)

5.3.5.1 Description

A streamlined timer: the zone watches for start signal, and a controllable interval later, it sends a signal. The delay can be randomized, and it can be paused, and reset. It always provides the remaining time (in seconds) on one output

ME INTEGRATION

delayFlag is intended for two main purposes:

- To introduce a controllable / randomizable delay between the input signal and output signal. Example: respond to a request after a few seconds to give more realistic appearance. Stack it on a zone and use zone-local flags for portability and minimal requirements.
- To act a timer that, after it runs down, sets a flag. Use it to set time limits in a mission or control response to user actions (if players do not achieve this within a time frame, do that). For this, use a separate zone, and global flags.

START/RESET, STOP

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.

Simply put “startDelay?” resets and then starts the timer. Note that the timer does not have to run down before it can be restarted. If this happens while the timer is running down, the timer is also re-started. Note that you can use this is a “dead man switch”: as long as when this signal is received before the timer runs down, it restarts and never sends the “delayDone!” output.

If “stopDelay?” triggers, the delay is stopped indefinitely until it is reset and restarted with a signal on “startDelay?”

PAUSE, CONTINUE

In addition to start and stop, you can pause and continue the delay time while it is running with the “pauseDelay?” and “continueDelay?” inputs. Pausing or continuing a timer that is not running has no effect.

REMAINING TIME

At any time, the remaining number of seconds on a timer is available on the “delayLeft” output. If the timer has run down, or hasn’t yet started, that number is -1

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

5.3.5.2 Dependencies
dcsCommon, cfxZones

5.3.5.3 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

5.3.5.4 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!	DML output. The flags to bang! after the delay has passed. Use only one synonym per zone
method delayMethod	DML method for outputs Defaults to “inc”
f? in? startDelay?	DML input Watchflag. When triggered starts the delay. Use any synonym, but only one per zone.
triggerMethod delayTriggerMethod	DML Method for input Watchflags Defaults to “change”
stopDelay?	DML input Watchflag. When triggered stops a running delay. Has no effect on a stopped delay Defaults to <none>
pauseDelay?	DML input Watchflag. When triggered pauses the delay to be continued later. Has no effect when triggered when the timer isn’t running or is already paused. Defaults to <none>
continueDelay?	DML input Watchflag. When triggered continues a paused delay. Has no effect when triggered when the timer isn’t running or not paused. Defaults to <none>
delayLeft#	Flag that carries the number of seconds left on the timer (including when delay timer is paused). Carries -1 when timer is not running. Defaults to <none>

5.3.5.5 Persistence

DelayFlags automatically supports persistence. If a delay flag was running at the point of saving the mission, the time delay is set to the remaining number of seconds when the mission loads data.

5.3.5.6 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.

5.3.6 raise Flag

5.3.6.1 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 flag methods. In conjunction with local flags, raiseFlag becomes a more zone-local replacement for many module's onStart option.

A raiseFlag module sends one signal (e.g. increment, decrement, flip, set a to a specific value etc.) on its output to connected flags when the mission starts, or after a delay. raiseFlag allows you to randomize the time, and you can prevent it from sending the output signal to connected flags if you stop if before its timer runs down by using the stopFlag? input. The latter allows you to easily design 'race against the clock' scenarios.

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

As of version 3, raiseFlags supports persistence, allowing you to set up timers that automatically run down correctly after a restart (i.e. if 20 minutes remained on the countdown clock, that interval is set correctly after re-load).

Note that raiseFlags triggers at most once. If you require a flag to be triggered more often or regularly, look at the 'pulseFlags' module.

5.3.6.2 Dependencies

dcsCommon, cfxZones

5.3.6.3 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

5.3.6.4 ME Attributes

Name	Description
raiseFlag!	Marks this as a flag raiser. The value of this attribute is the flag that is to be raised after a delay. **MANDATORY**
method	DML method to set output flags **Examples:**

Name	Description
	• inc - increment the flag specified in raiseFlag! • #5 – set the flag specified in raiseFlag! to the number 5 Defaults to 'inc' – the flag's value will be incremented by one
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. Defaults immediately after the mission starts up (0.5 seconds to allow other modules to set up)
stopFlag?	DML input Watchflag, only useful in conjunction with afterTime. When triggered and raiseFlag is still waiting for afterTime, raiseFlag is 'disarmed' and no flag will be raised in the future. Once stopped, raiseFlag cannot be re-started.
triggerMethod raiseTriggerMethod	DML input Watchflag method.
timeLeft#	Direct output that contains the time (in seconds) left before this timer runs down. Once that the timer runs down, there are no more updates to this output
verbose	If set to true, this zone returns debugging information

5.3.6.5 Using the module

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

Add a zone to the mission and add the 'raiseFlag! and other attributes to that zone.

5.3.7 xFlags (Flag Combining / Testing)

5.3.7.1 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

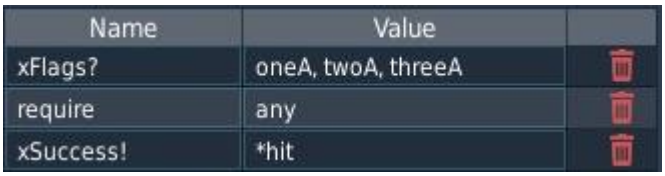
So, essentially, you'll break out xFlags whenever you need some decision testing, to see if enough of a mission's goals are fulfilled to warrant some action.

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".

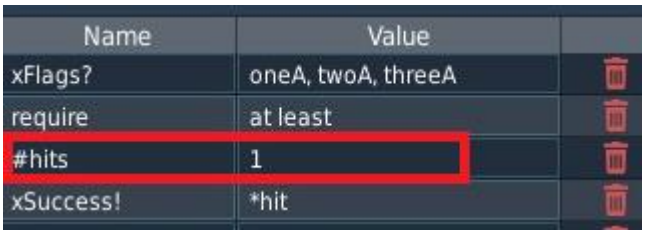
Currently, xFlags understands the following conditions:

- '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.

5.3.7.2 Dependencies
xFlags requires dcsCommon and cfxZones.

5.3.7.3 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

5.3.7.4 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	DML Input flag condition that must be met for each individual input flag. Is identical to trigger method for Watchflags except it is applied to each input flag individually. Defaults to "change"
xSuccess! out!	DML output. Names of flags to bang! when the evaluation of the 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!	DML output. Names of flag to bang! when xFlags detects a change in the input configuration. This werely detects a change in the input configuration, it has no relation to xSuccess!, except that (logically) any xSuccess is accompanied with a bang! on xChange! Defaults to <none>
xDirect#	Direct output. 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 flag's 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

5.3.7.5 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.

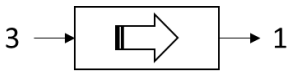
5.3.8 changer

5.3.8.1 Description

Changer is DML's "flag transmogrifier" and "gated switch". Whenever you need a flag's value transformed, or install a switch that controls when a flag's value may pass, you should turn to this changer module first

BASIC OPERATION: FLAG VALUE CONVERSION

Changer works by continually taking the input flag's value, changing it in some simple way, and then putting the result to the output flags. The way that the input value is changed to the output is controlled by attributes:



- `'inEval'`
  When set to true, the input value is interpreted according to the input watchflag's triggerChangeMethod. Otherwise, the input flag's value is read directly

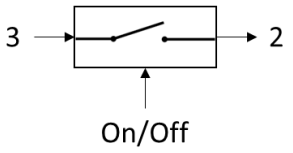
- `'changeTo'`
  Describes the 'operation' that is performed on the input signal to calculate the output. Changer supports the following:
  - `'bool'`
    Output value is 0 if input is 0, 1 otherwise (conversion to bool)
  - `'not'`
    Output value is 1 if input is 0, 0 otherwise
  - `'sign'`
    Output value is -1 if input is <0, 1 otherwise
  - `'abs'`
    Output value is the absolute of input value (e.g. outputs '3' for '-3' and '3')
  - `'negative'`
    Output value is input value multiplied by -1, turning the sign.
  - `'direct'`
    Output value is the same a input value. Used primarily with when changer is functioning as a gated switch

- `'Min'`
  When present, sets the lower limit of the output value, e.g., if Min is set to 6 and processed (see above) result is 3, the output value is set to 6

- `'Max'`
  When present, sets the upper limit of the output value, e.g. if Max is 2 and the processed (see above) result is 9, output is set to 2.

BASIC OPERATION: GATED SWITCH

Another central feature is that changer can be 'switched': when turned on, it takes the input value and (after processing) propagates it to the output. When you turn it off, input and output become separated, and the input values no longer propagate.



Whatever for? Often, you want a flag to trigger a module, but only under certain conditions. Phrased differently, you need something that can conditionally block a flag: We need a way to switch (turn on/off) the transmission of a flag to a module. This is what changer is designed to do. So, we propagate the input to output when the switch is 'on', and simply leave output as it is when switched off.

Changer sports two, sequential options to separate input and output:

- `changeOn?` and `changeOff?`
  These are (triggered) watchflags that are used to individually turn the Changer off (changeOff?) into paused mode, and on (changeOn?). When starting up, a Changer starts in 'On' state, unless you provide a 'changePaused' = 'true' attribute.
  When turned off (paused), the only way to turn a Changer back on is through a signal on changeOn?.

  Note that when you pause/turn a changer off, that state overrides any potential signals on the changeOn/Off? Input.

- `changeOn/Off? and NOT changeOn/Off?`
  Atypical for DML, this is an **immediate** (i.e. it does not trigger on change but works directly on the input value) input. For "changeOn/Off?", if the value of this flag is 0 (zero), the output flag is no longer changed. In all other cases, output is updated from input. Inversely, for "NOT changeOn/Off?", the gate opens when this flag's value is 0 (zero) and closes for any other value.

  Note:
  This is overridden when the changer is paused of turned off with changerOff? – in that case no signal is propagated to output.

NOTE
Changer's "immediate" switches "changeOn/Off?" and "NOT changeOn/Off?" do not wait for a change on input but derive the gate's state (open/closed) directly ("immediately") from the input value. They work considerably less refined compared to DML's Watchflags, but make controlling gates a lot simpler in many cases.

I recommend that you only use one method to control the switch – but it is possible (and required in some complex cases) to use both switch controls together.

DEFAULT OPERATION
Changers often work in both modes simultaneously: an input value is converted to an output value, while at the same time a different flag controls if the converted value flows to the output. Out of the box, a changer is configured as a simple pass-though:

- InEval: false, input is read directly
- changeTo: direct, input is passed to output unchanged
- no min, no max defined
- paused: false, gate is open

- on/off? and NOT on/off? disconnected

5.3.8.2 Dependencies

Changer requires dcsCommon, cfxZones

5.3.8.3 Module Configuration

To configure changer via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it “changerConfig” (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

5.3.8.4 ME Attributes

Name	**Description**
change?	The input flag whose value is used to create the output signal **MANDATORY**
out! changeOut!	The output flag.
triggerMethod triggerChangeMethod	Watchflag method that is used to interpret change? input when inEval is set to true Defaults to ‘change’
inEval	When set to true, the input flag change? is interpreted as a watchflag under triggerChangeMethod’s rules Defaults to false
changeTo to	Operation to apply to the input flag to create the ouput value. The following operations are defined: • `'bool'` Output value is 0 if input is 0, 1 otherwise (conversion to bool) • `'not'` Output value is 1 if input is 0, 0 otherwise • `'sign'` Output value is -1 if input is <0, 1 otherwise • `'abs'` Output value is the absolute of input value (e.g. outputs ‘3’ for ‘-3’ and ‘3’) • `'negative'` Output value is input value multiplied by -1, turning the sign. • `'direct'` Output value is the same a input value. Used primarily with when changer is functioning as a gated switch

Name	Description
	Defaults to “direct”
min	When defined, ensures that the output value has this value at minimum
max	When defined, ensures that the output value has this value. If max is less than min, output is always set to max
paused changePaused	When set to true, the changer starts in paused/off mode and the output flag is not changed. The only way to turn a paused changer on when off/paused is via a signal on the changeOn? input Defaults to false.
on? changeOn?	Turns a paused changer on, enabling transmission of the input signal (after processing) to output, opening the ‘gate’. Triggers on “change” Defaults to <none>
off? changeOff?	Turns a changer off, pausing it. Input signals are no longer processed nor propagated to output, closing the ‘gate’. Triggers on “change”. Defaults to <none>
On/Off? changeOn/Off?	Flag that when defined controls transmission of input to output when the changer in running (i.e. not paused). The value of this input controls the gate as follows: • 0: gate closed, no transmission • Anything else: gate open. Defaults to <none>
NOT On/Off? NOT changeOn/Off?	Works exactly like On/Off above, except that open/close values are exchanged: • 0 Gate Open • Anything else: gate closed Defaults to none. Be advised that if you also define On/Off? for the same zone, NOT On/Off will be ignored, and you receive a warning every second.

5.3.8.5 Using the module

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

To use, add the ‘change?’ attribute and required other attributes to a zone.

5.3.9 countDown

5.3.9.1 Description

Whenever you need to know that something has 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, *countDown* is here for you.

So – big picture: *countDown* counts the number of times that a command/signal was sent to the input.

BASIC FUNCTION

countDown is easy to understand: You set a limit (e.g. 5). The countDown zone then waits to receive signals/commands on the input "clock?". After countDown has received the same number of signals/commands as the limit, it send a command/signal to the output 'out!'.

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 outputs that make it so versatile:

- **T-Minus (Output)**
  Named after NASA's famous count down procedure, the "tMinus!" output simply send a command/signal each time that a signal/command is received on the "clock?" input *and the count is still above zero*. Hence, tMinus produces a signal/command every time *before* zero is reached, but not when zero is reached nor later.

- **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 send signals/commands every time that an input is received on 'clock?', and the counter is below zero, i.e. the count down has received more signals on the 'clock?' input than the start value (limit) specified, and is now 'below zero'. This **can't** happen if count down is set to loop

- **Value Randomization (Attribute)**
  The start value (limit) can 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 (limit) is calculated on each iteration of the loop.

- **Reset ability (Input)**
  At any time you can reset the count-down and it will strat over. If, for example, a faction needs to hold a zone for a duration, and they lose that zone, send a signal/command to *reset?* and the count-down is can reset.

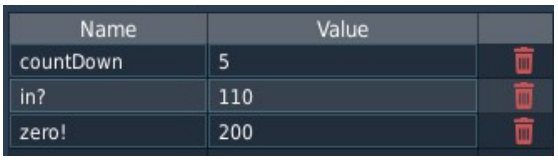
- **Enabling and disabling (Inputs)**
  A countdown can be enabled and disabled at any time. While disabled, any input signal on the clock? input is ignored.

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

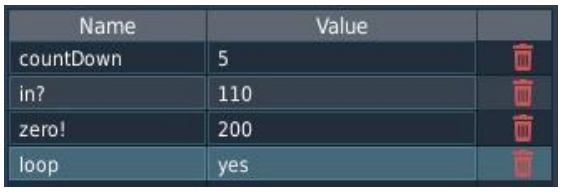
- **Classic Count-Down: ONCE AFTER SO MANY TIMES**

  In this configuration, you simply set the start value, and a signal/command is sent to zero! after the correct number of input signals were received on '*clock?*' (or '*in?*' as seen to the right)



- **EVERY NTH TIME – FREQUENCY DIVIDER**

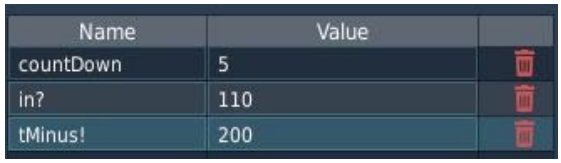
  This is a modification of above: the same set-up, except we have enabled 'loop'. After setting the start value (here 5), we now send a signal/command on zero! every n-th time a signal is received on in?/clock? (here every 5th time)



  Geeks call this configuration a 'frequency divider', as it divides the number of occurrences the signal received on of clock? by n.

- **ONLY UNTIL**

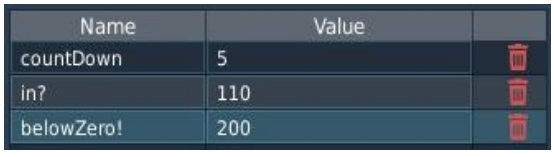
  This configuration uses a count down as a "limiter": it allows a signal on '*in?/clock*?' to trigger a signal on the output '*tMinus!*', but only until the maximum number of signals has been received on *clock?*. After that, commands to '*in?/clock?*' no longer result in signals sent from output '*tMinus!*'. So, in this configuration, only a maximum number of signals are sent out of '*tMinus!*'.



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

- **EVER AFTER / NOT BEFORE**

  This is the opposite of 'ONLY UNTIL': every time that the count-down receives a signal after the counter has counted to zero, a signal/command is sent from



'*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 command as '*belowZero!*'

5.3.9.2 Dependencies

countDown requires dcsCommon and cfxZones

5.3.9.3 Module Configuration

To configure countDown 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

5.3.9.4 ME Attributes

Name	**Description**
countDown	Marks this as a count down. The value of this attribute defines the number times that this module can be triggered until the count reaches zero ("limit") This value supports ranges: if you specify a range (e.g., "3-5") each time that the count down is initialized (at start, at reset, and when looping), a random number in the range (including upper and lower limit) is chosen as the start value. 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, a new random start value is taken from that range (including upper and lower limit)
method ctdwnMethod	DML Output method. Use only one synonym per zone. Defaults to "inc" (increment flags connected to output)
~~**count?**~~ in? clock?	Main input. Every time that a command/signal is received on this input, the count down proceeds down by one. Use only one synonym per zone. Defaults to <none>
triggerMethod ctdwnTriggerMethod	DML Input method that determines how inputs trigger. Defaults to "change"
zero! out!	DML output. The command/signal to send when the countdown reaches zero. Use only one synonym per zone. Defaults to <none>
tMinus!	DML output. The command/signal to send when count is lowered, and has not yet reached zero. Defaults to <none>
belowZero!	DML output. the command/signal to send every time that the count is

Name	Description
	decreased *and* is below zero. Note that if the 'loop' attribute is set, this can't happen. Defaults to <none>
counterOut#	Direct value output. Flags that are connected to this output are set to the current value of the count. The flags are updated each time that the counter is triggered via the clock?/in? input Defaults to <none>
disableCounter?	DML input. When triggered, this turns off (halts) the count down. Signals on the clock? Input are ignored while the counter is disabled Defaults to '<none>'
enableCounter?	DML input. When triggered, a disabled counter continues the count down. The count down picks up where it left off when it was disabled. Has no effect if the counter is enabled. Defaults to '<none>'
reset?	DML input. When triggered, it restarts the count down. If the count down is randomized, a new random start amount is selected. Defaults to <none>

5.3.9.5 Using the module

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

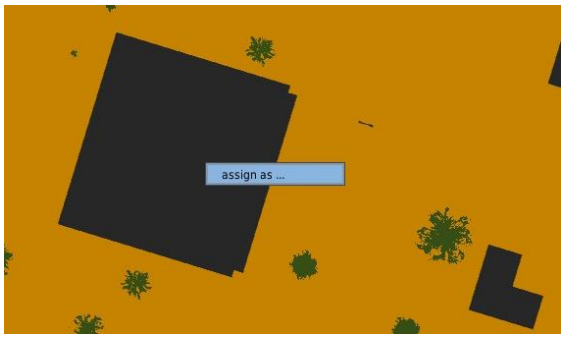
Add a zone to the mission and add the 'countDown' attribute to that zone. Add other attributes as required

5.3.10 objectDestructDetector (map/scenery object destruction)

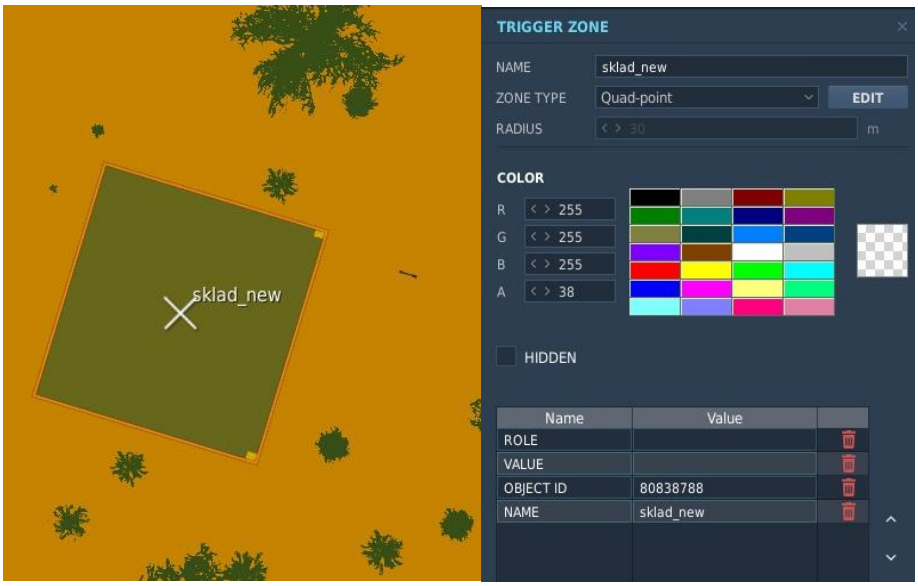
The object destruct detector (ODD) is a module that tightly integrates with ME and provide a convenient DML integration of DCS's ME condition "MAP OBJECT DEAD", giving you a simple method for your mission to detect and react to **destruction of scenery objects** (e.g. buildings, bridges, etc).-

5.3.10.1 Description

A peculiarity of ME is its ability to create a zone that exactly fits around a map object, and automatically assigns some attributes. This ability ostensibly was added in DCS version 2.7 for ME's (then new) "MAP OBJECT DEAD" condition and DML can expand on this ability in intelligent ways. 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, which shows the new zone that already has several attributes added:



By default, ME adds the attributes "Role", "Value", "OBJECT ID" and "Name", with attributes "Object ID" and "NAME" pre-filled by ME. DML's ObjectDestructDetector uses these attributes to work its magic.

When you include cfxObjectDestructDetector into your mission, it automatically seeks out all zones that have an "OBJECT ID" and "NAME attribute, and adds them to its watchlist.

Unlike plain vanilla ME, you can add the flags that you want to change directly on the trigger zone, and thus integrate them into your DML mission designs without having to resort to ME triggers or trigger conditions.

Integration with PlayerScore

NOTE: An unannounced, and probably unwanted change in DCS has rendered the below ability inactive. You currently cannot attribute kills of map objects to players. With some luck ED may change this in the future

DML's PlayerScore module has an automatic feature that can work together with ODD to allow you to set a score that is awarded to the player who destroys the map object. Integration is automatic. To tell PlayerScore how many points to award to a player, you simply add at least one of the following attributes

Name	**Description**
redScore	Score to award a red player for destroying this map object Defaults to <none>
blueScore	Score to award a blue player for destroying this map object. Defaults to <none>

There is no score for neutral players.

ME INTEGRATION

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

Name	**Value**	**Description**
f! destroyed! objectDestroyed!	name	DML flag to bang! when the scenery object is destroyed. Use any synonym but not both in the same zone

NOTE

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

All you need to do to have other modules react to a scenery object destruction is to connect the flag that is changed by the output (objectDestroyed!)

Note:

Earlier (pre version 2) versions of this module had used OBJECT ID instead of NAME. That occasionally required mission creators to update their missions when the map was updated. As of version 2 this is no longer required, DML's object destruct detector now simply works.

5.3.10.2 Dependencies

This module requires dcsCommon and cfxZones to be loaded

5.3.10.3 Module Configuration

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

5.3.10.4 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**
method oddMethod	DML Method to bang! the flags connected to output! when the map object is destroyed. Defaults to “inc”
f! destroyed! objectDestroyed!	The flag to bang! when the map object is destroyed. Use only one synonym per zone. Defaults to “*none”
redScore	Score to award a red player for destroying this map object Defaults to <none>
blueScore	Score to award a blue player for destroying this map object. Defaults to <none>

Note that the attributes OBJECT ID and NAME are automatically filled when you use the ASSIGN AS… right-click feature on a map object

5.3.10.5 Persistence

This module supports persistence in that it automatically preserves the status of the scenery object that it watches. If the object is destroyed, upon loading from file the object is removed, and the destroyed! Flag value is set to the same value that it had when the mission data was persisted.

5.3.10.6 Using the module

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

5.3.11 SpawnZones

5.3.11.1 Description

SpawnZones is a module that allows you to add Unit Spawners (i.e. points on the map where new ground units are created in-mission) to your mission. With a 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

Spawning logic per spawner is as follows:

- When the spawner is monitoring units (the last-spawned group) and that group has disappeared (by being destroyed, or otherwise removed from the spawner's oversight, e.g. they were picked up by HeloTroops), start a cooldown timer
- Determine if spawner should spawn:
    - The "*spawn?*" input has triggered
    - the cooldown timer has run down and
        - the spawner is not paused
        - the spawner belongs to the correct faction
        - the number of performed spawns does not exceed the maximum number of spawns
    - Mission is starting up and spawner is not paused
- If the spawner should perform spawning:
    - Assemble a group name based on the *nameBase* attribute. If no nameBase attribute is given, DML assigns automatically generates a safe nameBase for the spawner.
    - Create units as described in the *types* attribute, one unit per type. If no types are given, a single "Soldier M4" type is used. Assemble a unit name based on *nameBase* for each unit.
    - Arrange all new units as described in the 'formation' attribute. If no formation is given, the units are arranged in '*circle_out*'
    - Place the group inside the spawn zone
    - If *useDelicates* is set to an existing delictes zone (see Delictaes module), add all units to that Delicates Zone's inventory. These units explode when damaged slightly.
    - Unless the *autoRemove* attribute is set to true, place the spawned group under this spawner's oversight. Any previously monitored groups are forgotten.

As you can see, 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).

After objects are spawned, the SpawnZone keeps an eye on the spawned group. Once the group has disappeared from the game (by deleting/destroying them or picking them up), a new spawn cycle begins with initiating a cooldown, and then spawning all units after the cooldown has run out.

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.

Making spawns "delicate" or "brittle"
You can assign a Delicate Zone to the spawner with the *useDelicates* attribute. If you do so, the named Delicate Zone is used to assign delicate status to the spawns, making them explode when they receive a tiny amount of damage. See Delicates module for more info.

Controlling spawn names
You can control how spawned objects are named with the 'baseName' attribute. Unit and object names in DCS missions are special in that at any time there must be at maximum one unit or object with that name. If a spawner or cloner dynamically creates a new object or unit with a name that already exists in the mission, the existing unit/object is immediately removed.

DML spawners create names for newly spawned units/objects by taking the spawner's 'base name' and adding a spawner-individual count that increases with every unit/object created. Since DML automatically assigns safe base names by default, all spawners create safe unit/object names that cannot come in conflict with each other.

Mission designers can override DML's default behavior and provide their own *baseName* for spawns. In that case, the onus is on the designer to ensure that no name conflicts arise. Since in many cases it's desirable (and safe) to use the spawner's zone name as baseName, DML provides a special convenience shortcut "*" for the name attribute. In that case, the baseName for that spawner is set to the zone's name.

If you need tighter control over a spawned unit/object's name, consider using a clone zone and that zone's "nameScheme" or "identical" attributes.

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	Description
~~f?~~ spawn? spawnUnits?	Input. When the spawner receives a command or signal on this input, it triggers a spawn cycle. **Each time that a signal is received ("command heard"), a new group is spawned** Use only one synonym per zone
activate?	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

Name	Description
	spawner
pause?	When the spawner receives a signal or command on this input, the spawner's 'paused' setting is forced to 'true. Use this to 'pause' a spawner

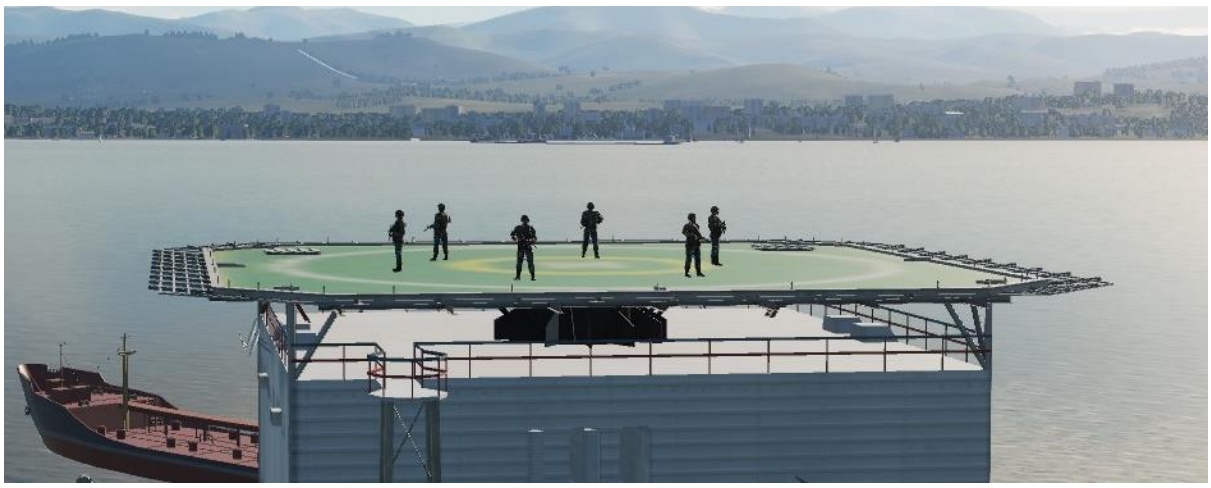
This simple mechanism allows mission designers to, for example, spawn troops whenever a player unit enters a zone (for very nasty surprises). Furthermore, spawners support watchflags that allow the mission to pause and un-pause (activate) spawners by changing a flag. This allows other modules (e.g. rndFlags)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



INTEGRATION WITH HELO TROOPS (requestable option)

You can use spawners and cloners (see below) to spawn units on-demand when a helicopter of the appropriate faction is close enough to the spawn zone. In order for Helo Troops to show the cloner for a player in the 'request' menu, the following must all be true:

- The player's helicopter must be within HeloTroop's 'requestRange' (500 meters by default) of the cloner/spawner (see HeloTroops)
- The cloner/spawner must have a 'requestable = true" attribute

- The cloner/spawner must match the coalition of the requesting aircraft. For cloners, this can be established most easily by using the "masterOwner" attribute and link to an owned zone (usually in conjunction with an airfield zone, or manually setting the owner flag (careful when using 'owned zones')
- The cloner/spawner must only clone/spawn units that are legal to carry by the helicopter (see HeloTroops)

5.3.11.2 Dependencies
Required: dcsCommon, cfxZones, commander, groundTroops

Optional: cfxHeloTroops

5.3.11.3 Module Configuration
This module does not need to be configured

5.3.11.4 ME Attributes

Name	Description
spawner	Marks this ME Zone as a spawn zone. The value for this attribute is a "type string" list that describes the ground units that are to be 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" **MANDATORY**
typeMult	A number. This is a multiplicator for the number of troops created. This will simply take above type string, and add n copies of the type string. Allows you to quickly create large groups with little effort Defaults to <none>
~~f2~~ spawn? spawnUnits?	Input (watchflag) to tell the spawner when to spawn. Each time that a signal/command is received on this input, a new spawn cycle is forced, ignoring all other settings like maxSpawn, cooldown, paused, etc. Defaults to <none> (no flag to observe) Use only one synonym per zone
pause?	Input. Each time that a signal/command is received/heard, the spawner's 'paused' setting is forced to 'true. Used to 'pause a spawner Defaults to <none>
activate?	Input. Each time that a signal/command is received/heard, the spawner's 'paused' setting is forced to 'false'. Used to 'activate' a paused spawner Defaults to <none>

Name	Description
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 ***Country***, not a Coalition for this attribute. 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). Defaults to 0
masterOwner	A string that references another ME Zone by name. It must match that Zone’s name. **A spawner only spawns *automatically* when the masterOwner’s owning faction is the same as the spawner’s country affiliation.** On the map, the spawner does not have to be inside the masterOwner’s zone, it can be hundreds of miles away. You can use this to start spawning reinforcements in a completely unrelated part of the map when units conquer the masterOwner zone. If no masterOwner is specified, the Spawner spawns as directed and disregards any surrounding zones that happen to be owned Defaults to <none>, spawn units as directed by “county”
baseName	A designation (e.g. “Hill Marines”) that is used to name units and groups during unit spawning (advanced uses only, usually safe to ignore). **If provided, you must ensure that resulting unit and group names are UNIQUE.** If you do not assign a base name, a safe baseName that is guaranteed to prevent possible name conflicts is generated. A **convenience shortcut “*”** replaces baseName with the name of the zone. This is a safe baseName and can be used for all spawns. If two spawners have the same baseName, spawned units from one spawner may lead to existing (previously spawned) units being removed from the mission if they have the same name. So if for some reason a spawner appears not to spawn units, make it a habit to check for potential name conflicts first. Default: <auto-generated safe baseName, based on zone name>.
cooldown	Time interval (in seconds) from when a new group can be produced after the last group was removed from the spawner. Defaults to 60 (seconds)
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.

Name	Description
	Defaults to 'false'
heading	The direction the spawned group it oriented to, from the center of the spawn zone. Defaults to 0
formation	The "formation" of the spawned group *when in assembles on the spawn location*. See dcsCommon for supported group formations. Note: this is not the formation that units assume to move to their objective. Defaults to 'circle_out'.
moveFormation	The formation that the units assume when moving to a destination. Legal values are (warning: case sensitive!) • `Cone` • `Rank` • `Diamond` • `Vee` • `EchelonR` • `EchelonL` • `Custom` Note that when units have attackOwnedZones oders, they ignore moveFormation and try to move over roads. Defaults to "Custom"
drivable	Controls whether a spawned unit can be controlled via from a player with the "Combined Arms" module. Note that some infantry units (e.g. Igla, Stinger) can also be 'drivable' although they are infantry, not vehicles. Defaults to false (unit is not player-controllable)
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 interfaces to groundTroops. See the main manual's *Orders* section for possible values. Spawners support the "wait" prefix (e.g. "wait-attackOwnedZone") to tell spawned troops that they should wait until picked up by a player-controlled helicopter (→heloTroops) before starting on their orders. Use "lase" if you want your spawned troops to laser-designate all targets within their range (see range, below) Defaults to "guard"
range	An attribute used to pass a range (in meters) value to some orders (e.g. JTAC lase range, detection/engage range). The range attribute has no effect on attackOwnedZone or captureHandHold. Defaults to 300 (meters)
target	An attribute used to pass a target (trigger) zone when used in conjunction with the 'attackZone' orders

Name	Description
	Defaults to <none>
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. Defaults to -1 (unlimited)
requestable	Interfaces with other modules (e.g. HeloTroops). 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. Defaults to false (cannot be requested by heloTroops)
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 spawner, you can use a single asterisk '*' as zone name. Supports a comma-separated list of trackers if you simultaneously want to pass the spawned groups to multiple trackers Defaults to <None>
useDelicates	Name of a Delicates Zone that is used to assign *delicate* (brittle, explodes when receiving minimal damage) status when this spawner spawns units You can use an asterisk ("*") as wildcard to refer to this zone Defaults to <none>

5.3.11.5 Persistence

Spawners fully support persistence. Note that since spawn zones can use DML-native routing via Orders, many persisted groups can resume their orders directly after loading a saved mission (meaning that they are not prone to the 'Waypoint 1' issue that cloned groups can have).

As with all other persisted groups, any unit that was destroyed will not be persisted, and any unit that was even barely alive will re-spawn fully repaired, fueled, and complete re-stocked with munitions.

5.3.11.6 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.

5.3.12 clone Zones

Clone Zones are among the most powerful DML tools. They are also among the easiest to use, so rejoice and read on! Clone Zones provide a quick, flexible, safe and easy-to-use method to spawn units in a mission (the pros call that 'dynamic' because it sounds important). Even better, Clone Zones use groups and objects that you place (and edit) with Mission Editor as their 'template' (i.e., you design what is cloned with ME), making integration and maintenance of clone zones in your mission easy and intuitive. Best of all, cloners can introduce a wide variety of randomization when they produce clones, so cloners will be one of your tools of choice when you are looking for ways to randomize your missions.

5.3.12.1 Description

Clone zones dynamically spawn units into the game while the mission runs. Instead of "type strings" that spawnZones use, these zones 'clone' (create exact copies of) groups that you place with Mission Editor. Cloners look at the groups that you tell them to clone as 'templates' for their clones. A template can be any group that you place on the map with Mission Editor: Simply create a group in ME, and then place a clone zone on top of it. It's that easy. When the mission runs, Clone Zones ("cloners") spawn new groups based on these templates.

Cloners are incredible flexible and provide your mission with abilities like

- Spawning groups and static objects dynamically on-demand (duh!)
- You can designate any group/static object (or collection of groups) as templates for cloners
- You can tell cloners to randomly choose from multiple templates
- When cloning groups, you can tell a cloner to randomize the location – of the group or individual units/objects.
- While placing clones on the map, you can tell a cloner to observe some map features (like placing units on roads, or to make sure not to place them too close to map objects like buildings)
- Cloners can be activated by signals (see '*clone?*' attribute)
- Cloners watch the last batch of clones that they created, and can create a signal on an output when the last batch is completely destroyed (see '*empty!*' attribute)
- Cloners can work together with other modules to imbue clones with certain properties (i.e. make them brittle or automatically be tracked)
- Clones can be made ti automatically despawn after a (randomizable) time interval (see '*depawnIn*' attribute)
- Clones can adhere to complex name schemes (advanced functionality, ignore at first)
- Cloners can despawn the last batch of clones on command (see '*wipe?*' attribute)
- Cloners can send out a signal when the last batch of clones has been damaged
- Cloners can output the health of the last batch of clones as a number on a flag

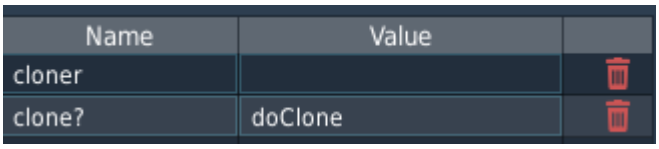
You usually use Clone Zones in combination with randomizers, pulsers and other modules to create clones whenever and wherever you need them.

5.3.12.1.1 How To – Clones Zones Quick Intro

Although Clone Zones offer more features than you can shake your keyboard at, setting up a basic cloner requires only a few easy steps:

- Create a group in Mission Editor
- Place a Trigger Zone above it, and name that trigger group
- Add the 'cloner' attribute to that zone

That's it. You now have a cloner active in your mission. However, it doesn't *do* anything, and to make matters worse, that group that you created in ME is gone. That's because a clone zone removes the originals to create the clones from. So, to use a cloner, you need one more step:

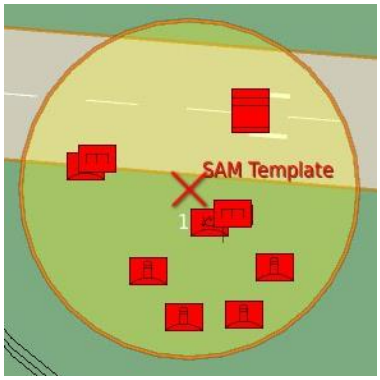


- add a 'clone?' input

Now, whenever the value on the flag that is connected to the 'clone?' input changes, the cloner creates a fresh copy of all units that are part of the template.

Note:
if you want a cloner to immediately spawn clones on mission start-up and can't be bothered to wire a flag to the 'clone?' input, set the 'onStart' attribute to true, and the cloner will run through a clone cycle when the mission starts.

5.3.12.1.2 Creating Templates



This is simple: Assemble a nice composition of units (they don't necessarily have to belong to the same group), then place a clone zone over at least one of the group's units.

At mission start, **any group that has 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. Important: that template *can also be used by other clone zones*.

IMPORTANT
Templates are created at mission start, and all units that are part of that template are removed. Later, when clones are created, they are always fresh copies from the template.

STOP!
Never set a group that you later wish to clone to **LATE ACTIVATION**. If you do, **that unit will span**, become late activated (vanishes), and – since the mission won't ever be able to find it – remain hidden and inactive forever.

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. Unless told otherwise, the template that a clone zone uses is its own.



What makes clone zones so powerful is that clone zones **can use other clone zones' 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
When, at mission start, a clone zone creates a template from the groups inside, all groups are removed from the game as part of the template creation process. You can respawn the template at mission start by adding an 'onStart = true' attribute.

Note that a cloner does **not** remove any units inside its zone if it uses other cloner's templates (i.e. it uses the "source" attribute). This is because such a cloner doesn't create templates itself so there are no units to remove for that coner.

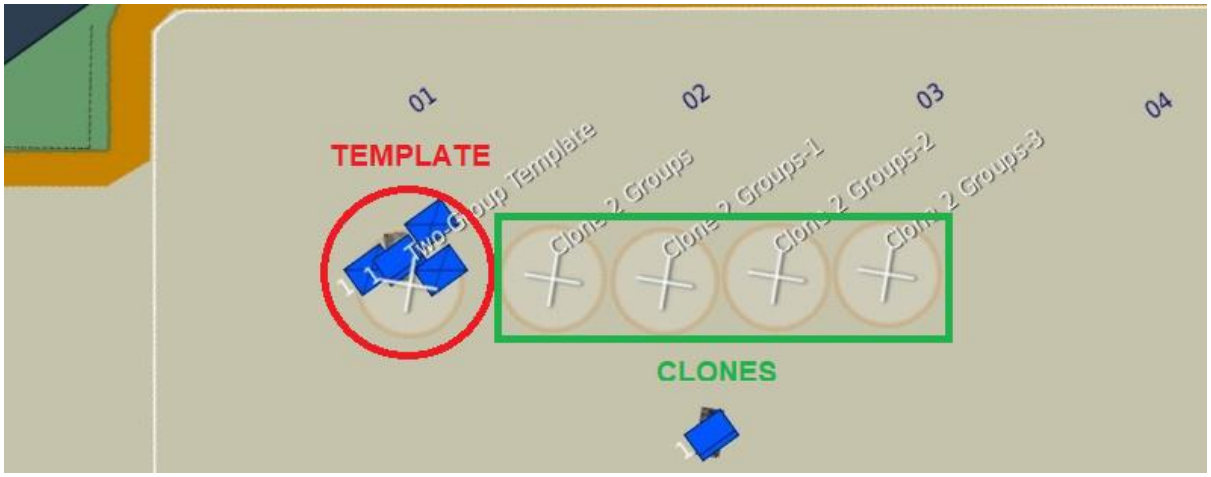
Clone Naming (default)
When a cloner spawns a new group, that clone's name is derived from the original group's name by taking the original name and adding a unique string. For example, if a group in a template is named "Dock Defenders", clones are named using "Dock defenders" as base, and have a new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

Note:
Clone Zones provide support for **custom unit names** via the "*nameScheme*" attribute. See "*Custom naming of cloned units*", later in this section.

Monitoring your clone's health
Cloners only remember that last batch of clones that they produced. That last batch, however, they remember fondly, and watch what happens to them. You can tap into that ability and receive signals when

- Units in the last batch are destroyed via the *damaged!* output
- The entire batch was destroyed via the *empty!* output
- The current percentage of living units in the last batch (100 means all units are still alive, 50 means half of them are dead, 0 mean that the entire batch is dead) via the *health#* direct output.

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)

Name	Value
cloner	two groups in template

"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 to 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).

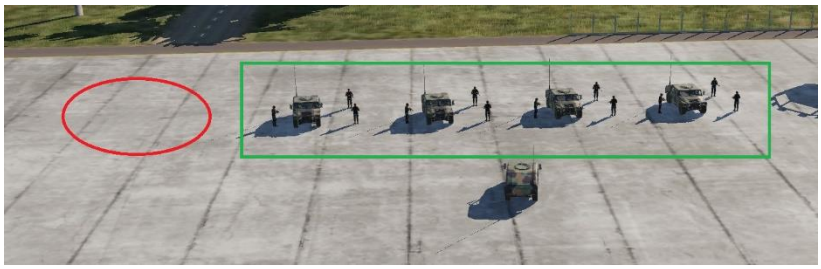
Name	Value
cloner	
source	Two-Group Template
onStart	yes

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.

Making clones "delicate" or "brittle"
You can assign a Delicate Zone to the cloner with the *useDelicates* attribute. If you do so, the named Delicate Zone is used to assign delicate status to all clones, making them explode when they receive a tiny amount of damage. See Delicates module for more info.

Randomizing a template's layout
By default, a clone zone create perfect copies from their original; all units/objects assemble exactly as you positioned them in ME (in relation to each other). You can change that: by using the target cloner's 'randomizeLoc' attribute: now the units are randomly distributed inside the clone zone (this also holds true for quad-shaped clone zones). There are several additional attributes that control distribution (they are described in greater detail later in this section:



- *wholeGroups*
  the group's formation remains intact, but the formation spawns randomly inside the zone. **Requires *rndLoc* to be true**.

- *onRoad*
  Each unit is moved to the nearest road (see image to the right). This is useful if you want to distribute all units of a group over the streets of an area with many roads. For a real-world example, look at the *"Street Sweep Tango"* mission that uses this feature.
  When used with the *wholeGroups* attribute set true, only the first unit is moved to the nearest road, the rest are assembled as the ME formation around unit 1.
  Can be used *without* rndLoc being true. Overrides *onPerimeter* and *inBuiltup*.



- *rndHeading*
  Randomizes each unit's initial heading. Can be used *without* rndLoc being true

- *onPerimeter*
  All units are assembled on the trigger zone's outline. **Requires *rndLoc* to be true.** Is not compatible with (it overrides) *inBuiltup.* Is overridden by *onRoad*

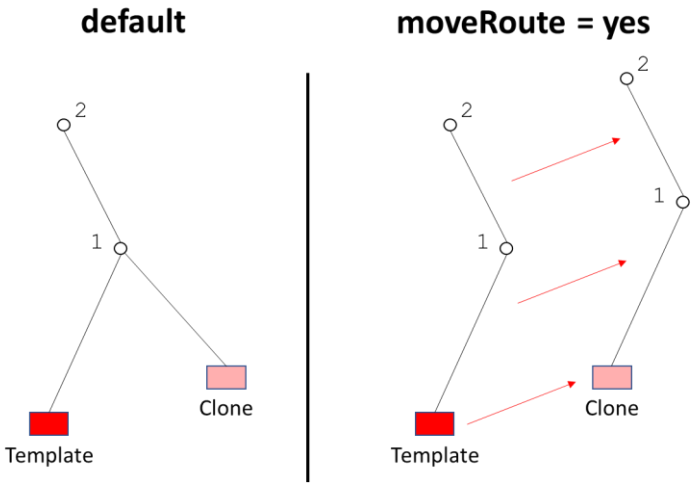
- *inBuiltup*
  This powerful attribute tells the cloner that the area where the unit(s) are to be spawned contains a lot of map objects (usually houses, like the slums of Cairo, or some densely populated warrens e.g. in Beirut) and that you want the units to be placed not closer than x meters to the nearest map object's center. For a real-world example, examine the "Angels of the Levant – Syria" mission, where closers are used with the inBuiltup option set to 15 to place evacuees inside densely populated Beirut.
  Requires *rndLoc* to be true. Is overridden when *onPerimeter* is set true. Is overridden by onRoad.



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:



5.3.12.1.3 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.

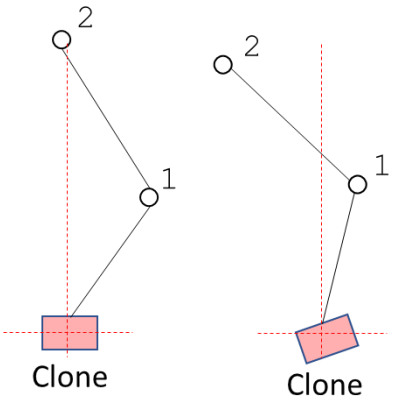


To allow you to do both, Clone Zones treat routes as follows:

- 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

Cloners that move (via linkedUnit attribute)

DCS supports zones that follow units via the 'linkedUnit' attribute (see "Core Abilities", all zones support this ability). Zones linked to a unit follow their master unit in tightly controllable ways. Clone Zones feature enhanced support for this feature:



- If you specify *useHeading* = *true* (see 'Core Abilities") the cloned zone's units are also rotated by the same amount that the master Unit turned relative to its heading in Mission Editor
- If you set *useHeading* (see "Core Abilities") **and** *moveRoute* (a Clone Zone attribute) to true, the group's entire route is also rotated the first unit's initial waypoint (see right). This means that you can use clone zone to align the entire route to the rotation of the master unit's heading (or rather: deviation from it's original heading. If the cloned group's route was designed to coincide with the master unit's heading, that route will always coincide with the master unit, no matter which way it turns.

5.3.12.1.4 Clone References

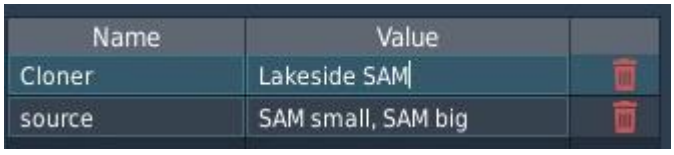
Some of the waypoint actions can reference other groups or units. For example, you can tell a group to escort or follow another group, or attack a unit. So what happens if you clone a group? There are a couple of possibilities:



- 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 part 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.

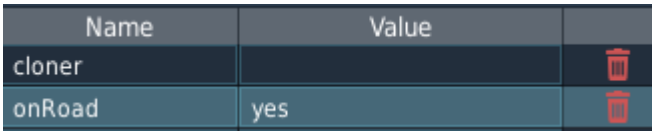
5.3.12.1.5 Using other zone's ('foreign') and multiple templates

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.



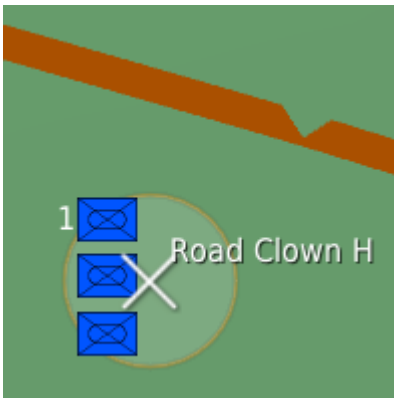
To further enhance randomization, clone zones natively support template randomization. When a clone zone supplies more than one template zones for cloning in their 'source' attribute, each time the zone undergoes a clone cycle, it randomly chooses one of the supplied templates to clone.

5.3.12.1.6 Spawning onto roads: onRoad Attribute



You can use the 'onRoad' attribute to force units to spawn on the nearest road to the intended spawn location (this option makes most sense when your cloner imports a template from another cloner, but can be used on any cloner). When you

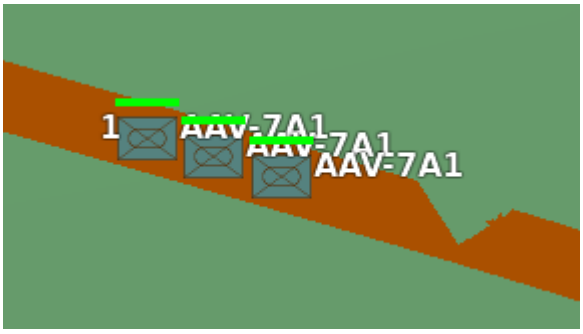
choose this option, the cloner will place any new spawn on the road nearest to the intended spawn location (after applying any randomization).



You can use this to great effect in densely populated regions (Cities) and in hilly regions to ensure that the units spawn on accessible surfaces (roads), not inside of buildings or steep inclines

The downside is that the next road may be far away, so make sure that if you activate the onRoad option, a road runs through the clone zone.

The cloner also attempts to keep minimal separation for each vehicle, but since this can't be algorithmically guaranteed to always work, will only try a couple of times before giving up and bunch the units together as it's told by DCS.



So, whenever you are using the onRoad option, keep the following in mind:

- If the nearest road is not inside the zone, the units will still move to the nearest road, and that may be miles away
- If there are multiple roads inside the clone zone, you can use randomization to distribute the units randomly over multiple streets. The onRoad option is applied after randomization, so the nearest point on the road can vary with each invocation.
- The original formation in the template will most likely be severely distorted
- The cloner takes at most 100 attempts to resolve separation between the units. If it can't resolve minimal separation, it gives up and bunches them on the road as close as they happen to be.

Using onRoad and rndLoc together with a city map allows you to quickly generate highly randomized mission scenarios. If you also supply multiple source templates to the spawner, you can create near endless variation with only a few simple attributes and zones.



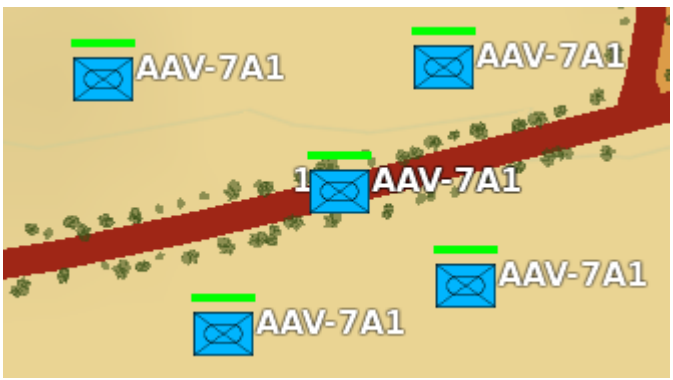
5.3.12.1.7 Randomly spawning inside cities / built up areas



If you want to randomize the spawn location of units inside map areas with lots of densely packed map objects (a.k.a “cities”) and prevent the units to spawn inside of buildings, you can use the inBuiltup attribute that tries to find a location that has enough clearing distance to map objects. For the Caucasus map, this is easy, and you can usually go with a value of 100 (meters). Other maps like “Sinai” or “Syria” may require a smaller value like 15. Using the ***inBuiltup*** attribute requires that you set ‘rndLoc’ to true. onPerimeter and onRoad must be false (they are false by default)

5.3.12.1.8 wholeGroups: randomizing groups versus units

The attributes *rndLoc*, *onRoad* or *rndHeading* affect units individually. This usually breaks a group’s formation (i.e. that careful arrangement that you made in ME). If you want to keep that arrangement, you can apply the ‘***wholeGroups’*** attribute which will use a group’s first unit, apply *rndLoc*, *onRoad* and *rndHeading* (whichever apply), and then arrange the rest of the group around that unit. For these attributes this means:



- *rndLoc*: the first unit is placed randomly. This means the entire group is moved to the randomized location, keeping the formation intact
- *rndHeading*: the entire group, while keeping relative formation, is rotated around unit 1
- *onRoad*: only unit 1 is guaranteed to be placed in the middle of a road. All other units

are arranged around it in the same formation as you set up in ME (see image to the right)
- *onPerimeter*: only unit 1 is placed on the perimeter, the rest is assembled around it.

5.3.12.1.9 Cloneable Units: 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 template that contains an oil tanker onto a mountain top.

Static Objects

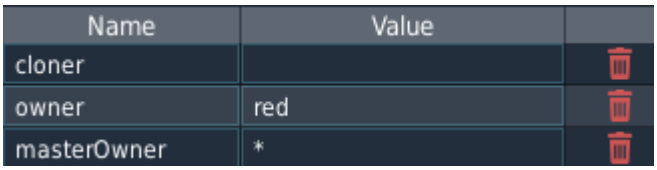
When you include static objects into a template, remember that a spawner's 'empty!' flag is only triggered if all units are destroyed – and this includes any static objects spawned.

Cargo (Static Objects)

When you clone static objects that can be used as cargo (by having the relevant checkbox ticked), any cloned such object is also cargo. If the CargoManager module is installed, a cloneZone automatically hands off cloned cargo objects to it for management.

5.3.12.1.10 Cloned unit coalition

By default, units that spawn belong to the faction that was given in the **template** (if a template contains multiple groups from different factions, the



cloned units will as well. This often results in cloned units that instantly engage each other). So, no matter what faction the clone zone belongs to (by default, zones in DML are neutral), the spawned units match ownership of those in the template.

You can, however, override ownership of the cloned units by specifying a *masterOwner*. All spawned clones then belong to the coalition that owns the zone indicated by masterOwner. Since it is common to use the clone zone's ownership as the master owner, a **convenience shortcut "*"** is available to set the clone's own zone as master owner (often used when the cloner imports templates from other factions, or if the clone zone can be captured)

NOTE:
Clone zone's 'masterOwner' behavior is markedly different from how *masterOwner* works in spawnZones. There it is used to suppress automatic spawning. In cloneZones, this attribute controls the spawning faction instead. Caution is advised.

*5.3.12.2 Custom naming of cloned units (*nameScheme *attribute) [Advanced Topic]*
In DCS, all units and groups must have unique names (i.e., the same name must at most exist once at the same time). If you spawn a unit or group with a name that already exists in the game, the existing unit/group is removed. Therefore, all cloners default to a safe naming scheme that guarantees that all cloned units and groups get along with each other: although they are named similarly to their 'original', they never match another clone's or existing group's/unit's name.

Default unit naming behavior
DML does this by taking the original name, then adding a hyphen ("-") and then a unique number based on DML's "unique ID" function. Therefore, if a group's original was named "Batumi Defenders", a clone's group name could be "Batumi Defenders-76592".

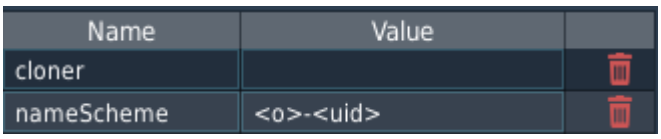
Unless you explicitly change this by adding specific attributes to a clone zone, above is how names are assigned to cloned units, and this works well for over 99% of all use cases.

Optional: Assigning your own clone name schemes
Occasionally, you need to control the name of cloned units. This is usually the case in

- when you want your units to have **better, more informational names** that replace the functional names that are handed out by default. You can, for example, have the cloned units include the spawn zone's and the template zone's name or some other information that you may think helpful/pertinent for your mission
- if some other part of your mission relies on the **specific name of unit** (for example when a moving zone is linked to a unit), and/or you want to **replace existing units** (of the same name) with new ones, even across multiple cloners, across the entire map.

To provide this ability, cloners offer *optional* attributes to tightly control how names are assigned to units:

- *nameScheme / groupScheme* with these attributes, you provide your own "naming scheme" that uses wildcards (similar to messenger) to provide a tightly controlled method to assemble unit/group



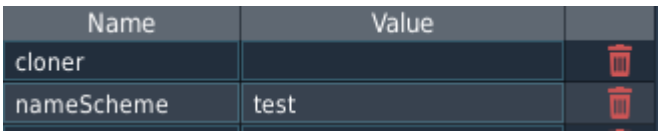
names. The value field in the *nameScheme* and *groupScheme* attributes describe how the name is put together. You supply a string that contains 'wildcards' enclosed in angle brackets ("<>"). At runtime, the name is assembled from this description and the wildcards are replaced with their current values (see below).

- *nameScheme* and *groupScheme* Wildcards
  - `<o>`
    original name of the unit (the unit's name in the template) or group
  - `<z>`
    name of the clone zone (note: this is the name of the cloner that is spawning)
  - `<s>`
    name of the zone that the source template is from. If this cloner does not use another cloner's template (via source attribute), it is the same as <z>
  - `<uid>`
    assigns a unique number from DML's global unique number generator, e.g., "76533". This number is unique across the entirety of DML. Its value cannot be predicted.
  - `<i>`
    assigns a unique group-internal count number. When a new group clone is started, this count starts at 1. Each time <i> is used in a name, it is increased by one. For group names, this is always "1"
  - `<lcl>`
    assigns a unique count based on this zone's unique count. A zone's local unique count is initialized at mission start according to the config settings (usually 1). Each time <lcl> is used, this zone's unique count is increased by one. This count is *zone-specific*, and can differ between clone zones on the map. This count is shared between group and unit names if your use both groupScheme and nameScheme in the same zone.
  - `<g>`
    assigns a unique count based on cloneZone's unique count. CloneZone's unique count is initialized at mission start according to the config settings (usually 1). Each time <g> is used (from any clone zone), cloneZone's global count is increased by one. This count is shared between all clone zones, and therefore unique for all cloners

When you use your own naming scheme, it is important to remember that if you spawn a new group or unit, and a group/unit with that same name already exists, the existing one is removed. You can use this to intentionally replace groups/units. If you want to avoid replacing existing units, you should ensure that your naming scheme is 'safe', i.e., it will produce unique names across the entire mission. By default, cloners use safe naming schemes for both units and groups.

Clone Zones auto-enforce *in-group unique* names:
Note that there currently are no precautions nor validations built into DCS's group spawn method to prevent a mission from spawning a group with multiple units that share the same name. If you do that, you will end up with multiple units that can't be easily accessed any

more because DCS's assumption that all unit names are unique no longer holds true. One of those same-name units remains accessible, while the rest become inaccessible until the mission ends.



Theoretically, DML's name schemes allow you to create schemes where multiple units inside the group share the same name, which can be problematic. The example on the right creates the same name 'test' for *all* units that it clones. If they were to spawn into the mission, the results can be unpredictable.

To avoid a situation where there are multiple units of the same name in the mission, **DML**'s cloners **enforce in-group unique names**. Should a name schema lead to multiple units sharing the same name, clone zone will ensure each unit's name uniqueness by adding the letter "`x`" to the generated name until it becomes unique inside the group.

If that zone's verbosity is enabled, you'll also receive a warning every time that clone zone must enforce a unique name, and what that unique name is:



Safe Naming Examples:

`<o>-<uid>` – this is what each cloner uses by default (when no name scheme is given) for units and groups: the original group/unit name plus a hyphen '-', plus DML's unique number function (e.g. 76546). Example "Batumi Armor-1-76546". This naming scheme is safe to use across all cloners as it can't create duplicate names (the <uid> part ensures that) even if you use nameScheme and groupScheme for the same cloner.

`<z>:<o>:<uid>` – A verbose name that concatenates the spawning zone's name, a colon, the original group/unit name, another colon and a unique number. Example "Senaki SAM:Zoo-1:78812". This name is safe to use across all cloners, and has the advantage that it informs you which cloner spawned it

`Clone <s>@<z>|<lcl>` – A verbose name that concatenates "Clone ", the template's zone name, an at symbol '@', the spawning zone name, a pipe symbol, and a zone-local count. Example: "Clone NASAM-T1@Senaki SAM|12". This name is safe to use across all cloners as any combination of <z> (each zone has a unique name) with <lcl> (a unique number inside that zone) in the schema guarantees unique names across the map.

`<z>::<g>` – This concatenates the spawning zone's name with "::" and a unique global number across all spawners. Example: "SAM Kobuleti::21". This name is safe to use across all cloners because zone names must be unique, and the global count is also unique across all cloned units/objects

`<o>-<g>` – The original unit name, plus a hyphen '-', plus a unique global count. Example "Batumi Armor-1-22". Safe, since <g> is guaranteed to be unique across all clones.

Unsafe Examples:

`T-90 boss` – since the scheme contains no wildcards, **all** units would be made to spawn with that exact name "T-90 boss". Clone Zone will ensure that this does not happen within

the group itself (it makes all unit names unique within the group by adding “x” until it becomes unique). Furthermore, if the cloner runs another cycle, all previously spawned units/objects are replaced with fresh clones of the same name.

This naming scheme *can* make sense with one-unit groups that then makes this one-unit spawn in different locations of the map, always with the same name “T-90 boss”. See also the section on the *identical* attribute for an alternative to this effect.

`<s>-<lcl>` – concatenates the source template’s zone with the local zone-unique count. This is unsafe because if other zones use the same schema and source, this can lead to non-unique names – a very difficult to debug situation.

`<z><i>` - concatenates the cloner’s zone name with a group-internal unique count. This is unsafe in multiple ways:

- Internally: if the template contains more than one group, all units/objects in the second or later groups replace the one from the first because their names are all identical.
- Globally: When the same spawner spawns again, all previously spawned units are replaced with new spawns of the same name.

Special Scheme Examples

`<o>-<uid>` – this same naming scheme is used by all clone zones if you do not specify your own scheme for that zone.

`<o>` – this special effect scheme copies all unit names from the original template (Note: this is different from using the *identical* attribute, as groups and units still receive unique IDs. See the section on the *identical* attribute, below for details). As a result, it replaces all previously cloned units from that template with that scheme. It can be used to ‘replenish’ and/or ‘teleport’ units across the map. Since it uses the template’s original names, it is compatible with many scripts that access units by *name* (but it is *not* compatible with scripts that access units by ID. Use the ‘identical=true’ attribute for that)

5.3.12.3 [Advanced Topic] Identical: there can only be one *– but it* can *be different*
For some advanced mission effects (for example AI units that can switch sides), you may want to create a clone of a template with its original names and IDs as defined by the template. This is desirable in rare cases and can be a powerful tool to teleport, reset AI units on-demand (fleet tankers, AWACS etc., on long-running missions) or even have units switch sides.

“Identical” clones **use the original template’s names and IDs for both groups and units**: when they spawn, they replace any prior existing copy. Due to the fact that their name and ID are defined in ME, they are fully compatible with any script or ME function that accesses units by name or ID (the latter being impossible when cloning with name schemes).

Remember that only one ‘identical’ cloned template can exist in the mission at any time, even if you use different cloners to spawn the identical template (it is this fact that makes it so desirable for special effects).

Important:
'identical' **only applies to the clone's names and IDs**. It does not apply to other attributes. This means that the resulting cloned units **can** be **different** from the original template in many ways: they can belong to other coalitions, spawn at different locations, and have their routes changed.

Note
The identical attribute and name schemes are mutually exclusive. 'Identical' overrides any 'nameScheme' / 'groupScheme' attribute and produces a warning for that clone zone.

Note:
Currently a strange quirk in DCS can intermittently fail to assign the correct Group ID for 'identical=true' when it spawns Aircraft groups that replace already existing ones. DML detects this and will correct this issue by re-spawning the group one second later. If that fails as well, DML continues to re-spawn the group until it succeeds. This process can take up to three seconds and may cause minor visual inconsistencies ('blinking' of the affected aircraft).

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

- The clone zones local (<lcl> wildcard for name scheme) is initialized from cloneZones' *localCount* config setting (usually 1)
- If no 'source' attribute is present, the clone zone looks for any unit (air, ground, sea) that is inside the zone, and copies all groups for which at least one unit is inside the zone's boundaries into a new template. All groups that are part of this zone's template are removed from the game.
- If a "source" attribute is present, no units are removed.
- If neither "source" attribute nor groups are found, a warning is given that no template was created.
- if there is an 'onStart = true' attribute present, a clone cycle is initiated.

5.3.12.5 What happens during a Clone Cycle
Each time a clone cycle starts,

- If the 'preWipe' attribute for the zone is true, any surviving groups/units of the previous clone cycle are removed from the game. Additionally, to prevent 'stacking' issues with units (a result from how DCS removes static units), spawning of clones is delayed 0.5 seconds to allow the game to realize that the previous units have really gone.
- If the 'declutter' attribute of the zone is true, the volume of the clone zone is cleared of debris, such as burned-out husks. Note that is a wreck is still burning, that wreck is removed, but the flames will remain for up to a few minutes. These flames are a visual artifact from DCS, and do not harm units even if entirely engulfed in them.
- 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.

- If no source attribute is given, the clone zone's own template is loaded. It's currently impossible for a clone zone to have its own template and use other cloner's templates via 'source'
- The template is used to clone of all groups that are in the template.
- While cloning groups, the cloner
    - Updates all locations (to account for the offset between source template and spawning cloner)
    - Randomizes locations and heading if so desired (rndLoc, onRoad), rndHeading)
    - updates all routes according to *moveRoute*
- Before spawning, groups, units and static objects are assigned ID and names as follows:
    - In the rare, advanced use case that the clone zone has an "identical=true" attribute, all **units and groups receive the same ID and name from the template**. Spawning such a clone result in special effects as described in the section "Identical: there can only be one".
    - Otherwise
        - all ***Groups***, ***Units*** and ***Static Objects*** are issued **a *unique ID*** based on CloneZone's *uniqueCount* number base (that you can set with cloneZone's config zone)

        - **Groups** are issued a **Unique Name** based on the groupScheme (if present) or the group's original name and DML's unique number base (no groupScheme). For example, if the original (template) group name was 'Senaki Armor', the cloned group's name could be 'Senaki Armor-76577

        - **Static Objects** and **Units** are issued **a Name** based on the zone's **nameScheme** attribute.
            - If you do not supply a name scheme for the cloner, the unit's name is created based on the unit's name and DML's unique number base. For example, if the original unit's name was "T90 Tower", the cloned unit's name could be 'T90 Tower-76671'
            - If there is a name scheme for the clone zone, a new name is assembled for each unit for every group, according to the name scheme.
                - If the name scheme results in a duplicate name *within the currently assembling group*, this conflict is resolved by modifying the created name to make it group-internally unique.
                - If the name scheme results in a name for a unit that is already known to the mission, the clones unit will replace the existing one. In this case, the cloner fetches the existing unit's ID and assigns it to the clone so that a complete name and ID match-up is achieved.

- After all group/unit ID and names are assigned, the cloner resolves internal and external references for all tasks. That way, for example, a group cloned Eagles can find and escort the correct cloned Tanker.
- If the clone zone's local, or CloneZone's global verbosity is enabled, and the cloner uses *nameScheme*, a name pre-spawning name-validation is run. This warns you when the imminent spawn will result in existing groups or units/static objects will be replaced by the spawning clones. This validation will also detect clone-internal cross-group replacements (i.e. if the template contains more than one group, and the name scheme resulted in duplicate names across two or more groups). The validation run only produces warnings and will not prevent replacements from happening
- First all Groups/Units in a template are spawned, followed by all static objects
- While spawning, a cloned unit's/static object's faction is determined as follows:
    - If no masterOwner attribute is set, they spawn with the same faction that was given in the template.
    - If a zone is given in the *masterOwner* zone is given, the cloner first determines the master zone's currently owning faction, and then selects a country from that faction according to the mission's coalition setup.
      All units/static objects spawn as members from that country.
- After spawning,
    - If the nameScheme or identical attributes are set, verify that the assigned target ID for units and groups are kept (a bug in DCS core currently may temporarily cause an ID mismatch). If the IDs do not match up, the cloner deletes and respawns any mismatching group until the IDs are matched.
    - if the cloner's *useDelicates* attribute is set, it hands off all spawned units and objects to that delicate zone for tracking.

5.3.12.6 Referencing Units/Objects that are used in Templates ("identical" exception)
All template's original groups (and their units/static objects) are destroyed when the template is created at mission start. This means that normally, any units or static objects that are part of a template *should not* be used in **ME's trigger conditions** (e.g., 'UNIT IN ZONE' or 'GROUP IN ZONE'): they no longer exist in the game – they were destroyed at mission start.

There is an important exception to above rule: any units that are part of a template that is spawned with a clone zone that has the '**identical**=**true**' attribute *can* be used in such references, because they can re-spawn into the game with their correct ID and name intact. Note that since all original templates are destroyed at mission start, this means that the unit must be actively spawned at least once.

IMPORTANT:
Units/Static objects that are part of a template *can* reference other units/static objects that are also part of a template from ME's **Waypoint Actions** (for example a tanker and their escort group). This is possible and works well because Clone Zones resolve cloned dependencies during the clone cycle, and can correctly connect actions to spawned units (see the section on 'Clone References' above).

5.3.12.7 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? ~~**wipe?**~~	Name	Observes 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. **Please note that the wipe? attribute is deprecated as it conflicts with the "Wiper" module's key attribute**

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 fully support AI 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

INTEGRATION WITH HELO TROOPS (requestable option)
Similar to spawn zones, you can use cloners to spawn (here: clone) units on-demand when a helicopter of the appropriate faction is close enough to the clone zone. In order for Helo Troops to show the cloner for a player in the 'request' menu, the following must all be true:

- The player's helicopter must be within HeloTroop's 'requestRange' (500 meters by default) of the cloner/spawner (see HeloTroops)
- The cloner/spawner must have a 'requestable = true" attribute
- The cloner/spawner must match the coalition of the requesting aircraft. For cloners, this can be established most easily by using the "masterOwner" attribute and link to

an owned zone (usually in conjunction with an airfield zone, or manually setting the owner flag (careful when using ‘owned zones’)
- The cloner/spawner must only clone/spawn units that are legal to carry by the helicopter (see HeloTroops)

5.3.12.8 Dependencies
Clone Zones requires dcsCommon, cfxMX and cfxZones.

5.3.12.9 Module Configuration
To configure cloneZones via a configuration zone,

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

Name	**Description**
verbose	Show debugging information. Default is off
uniqueCounter	The starting number for cloneZones global ‘unique’ number generator, which is automatically increased by one (1) each time it is used. Defaults to 9200000
lclUniqueCounter	The starting number for each cloner’s own ‘unique’ number generator, which is automatically increased by one (1) each time it is used. Defaults to 1

5.3.12.10 ME Attributes

Name	**Description**
cloner	Marks this ME Zone as a clone zone. The **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 a 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, and no units are removed at start. 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 randomly picks a template from that list. 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

Name	Description
	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. When the clone zone is using *linkedUnit* (moving DML zone, a Core Attribute of all DML zones) in conjunction with *useHeading* (another Core Attribute), the entire route is also rotated to coincide with the linked master unit's heading differential to its original ME heading. 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.**
masterOwner	If **not** present, all *cloned units retain ownership from the template*. This can be problematic if you need to change the ownership of the spawned units (for example when you import templates via source, or the clone zone can be captured). If present, all cloned units are spawned for the faction (red/blue/grey) **that owns the zone named masterOwner**. There's a convenient shortcut to set the masterOwner to this clone zone (see below) **CONVENIENCE SHORTCUT:** When the master owner is set to "*****" (Asterisk) wildcard, it is set to the same zone as the cloner. This is a convenience shortcut for cloners that import foreign templates and convert them to their own faction, and to enable a cloner that can be conquered to spawn its own template according to the faction that owns the spawning cloner. Note that when you use the masterOwner attribute, the coalition for which this cloner spawns units can change with the ownership of masterOwner. This can be used with cloners that import templates to easily control spawning the same template that aligns to different factions by setting the spawning cloner's 'owner' to a faction, set masterOwner to itself 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.
despawnIn	If you add this attribute, all clones created by this cloner will be automatically de-spawned (removed from the mission) after the amount of seconds that you specified here.

Name	Description
	If the clones were already destroyed before the time runs out, nothing happens. *despawnIn* is compatible with the *empty!* output (meaning: if the last batch is auto-destroyed due to *despawnIn*, that will create the appropriate output signal. despawnIn supports a single number and range. If you supply a range (e.g. “600-1200”), each group (but not unit inside the group) or static object is assigned their own despawn time. Defaults to: <none> (no automatic despawn)
triggerMethod cloneTriggerMethod	Watchflag method to tell what to look for in inputs. Defaults to “change”
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. If units are preWiped, removing the remaining units happens immediately, while spawning of new units is delayed for 0.5 seconds to allow DCS to internally resolve ground height and prevent spawned new units to ‘fall to the ground’ or spawn suspended in mid-air. Default: false (no pre-wipe)
declutter	If this attribute is set to true, all debris and wrecks inside the clone zone are removed before a clone cycle is started. Use this to remove debris from destroyed units before re-spawning units in the same zone. Default: false (no declutter)
empty!	The value of this flag is changed according to *cloneMethod* when all units from the last spawn have been destroyed, including all static objects. **Note:** There are other outputs (for example damaged! and health#) that you can use to monitor the health of the last batch of clones and can get alerted if one of more units in your last clone batch get destroyed. See those attributes below. Defaults to <none>
method cloneMethod	DML Method for output flags Defaults to “inc”
deSpawn? deClone? ~~wipe?~~	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! will **not** trigger subsequently. **Please note that the wipe? attribute is deprecated as it conflicts with the “Wiper” module’s key attribute.**

Name	Description
	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. Defaults to <none>
useDelicates	Name of a Delictates Zone that is used to assign delicate status when this spawner spawns objects. As with the *trackWith:* attribute, you can use "*" to refer to this zone. Defaults to <none>
randomizedLoc rndLoc	Upon cloning, each cloned unit/object is placed randomly inside the zone. Works with moving zones. Works with quad-based zones. Defaults to false (use template's placements)
rndHeading	Upon cloning, each cloned unit/object assumes a random heading. Defaults to false (keep template's heading)
triggerMethod cloneTriggerMethod	Watchflag method for input flags. Defaults to 'change'
onRoad	Set to true to have any cloned unit deploy on the nearest road to the location it would spawn otherwise. This is very likely to change the template's formation, and can be a long way from the clone zone's intended spawn location, so use carefully. May also have other effects, such as bunching up the spawned units in unrealistically close proximity to each other. If set to true, it overrides whatever settings are used for onPerimeter and inBuiltup. Defaults to false (keep template's position)
wholeGroups	A modifier for the rndHeading, onRoad and rndLoc attributes. When set to true the clones are affected as follows: • rndLoc - randomizes the location of unit 1 of the group, and keeps all other vehicles in formation relative to that group. In fact, it only randomizes the group's position, not individual units • onRoad - now only the first unit is placed on the center of a road, all other units remain in location • rndHeading - with centerOnly, all units rotate randomly around unit 1's location Defaults to false (all units individually)
onPerimeter	**A modifier for rndLoc.** If this attribute is set to true, the units are distributed along the edge of the trigger zone. Can be used alongside *wholeGroups*. Is overridden when *onRoads* is set to true.

Name	Description
	When set to true, overrides any setting of *inBuiltup*. Defaults to false (keep template's positions).
inBuiltup	**A modifier for rndLoc.** If set, the units are placed on the map with at least as much clarence (in meters) to the center of any map objects in the vicinity. Use this when you want to spawn units on a free spot in map regions that have lots of houses like cities (hence the 'builtUp' name). **Note:** There is no guarantee that DML can find a sufficiently open (free) space inside the destination trigger zone. For example, some densely populated areas on the "Syria" or "Sinai" map are so densely packed with objects that it is impossible to quickly find a free space that is 20 meters wide. After several tries, it simply gives up and places the unit randomly without guarantee that the inBuiltup constraint (clearance) is kept. **Recommendation**: use 15 (meters = 45 feet). If that works well, you can try and decrease. Is overridden if either *onRoads* or *onPerimeter* is set to true. Defaults to <none>
nameScheme	By default, a clone zone uses a simple name scheme to ensure that all cloned units have a mission-unique name. The default name scheme is the original unit's name (as taken from the template), plus a hyphen "-", plus a DML-unique number (e.g. "76791") In certain situations, mission designers need or want a tighter control over how cloners name units: for better (easier to debug) "speaking" unit names, to achieve special effects (intentionally replace existing units), or to be able to interface with more predictably named units (e.g. linkedUnit attribute for moving zones) or other scripts. When you are using a name scheme for a clone zone, the onus to ensure that you don't accidentally replace an existing unit of the same name is on you. See 'Custom naming of cloned units' in this section for details. Clone Zone's name schemes supports wildcards that are filled in at runtime as follows: • <o> the original (as defined in the template) name of the unit • <z> name of the spawning cloner's zone • <s> name of clone zone to whom the template belongs that is currently spawning. If the cloner does not use a *source* attribute, it is the same as <z> • <uid> creates a DML-wide unique number. It is guaranteed to be

Name	Description
	unique across all clone zones. • <i> creates a group-local unique number valid only for this spawn cycle. It increases by one (1) with each use, and reset to 1 whenever a new group is cloned • <lcl> creates a zone-local unique number. It increases by one (1) with each use (unit or groups). This number is not shared among other clone zones, so it is not guaranteed that this number is unique across multiple cloner zones. • <g> creates a module-local unique number. It is increased by one (1) with each use. This number is guaranteed to be unique across all clone zones. Defaults to <none> (this means that clone zones uses its default naming scheme which is equivalent to <o>-<uid>
groupScheme	Like for units, a clone zone uses a simple name scheme to ensure that all cloned **groups** have a mission-unique name. The default name scheme is the original group's name (as taken from the template), plus a hyphen "-", plus a DML-unique number (e.g. "76791") Like for units, you can change the way how the cloner names groups (this is always independent from how it names units) Clone Zone's name schemes supports wildcards that are filled in at runtime as follows: • <o> the original (as defined in the template) name of the unit • <z> name of the spawning cloner's zone • <s> name of clone zone to whom the template belongs that is currently spawning. If the cloner does not use a *source* attribute, it is the same as <z> • <uid> creates a DML-wide unique number. It is guaranteed to be unique across all clone zones. • <i> always returns "1" • <lcl> creates a zone-local unique number. It increases by one (1) with each use (units or groups). This number is not shared among other clone zones, so it is not guaranteed that this number is unique across multiple cloner zones. • <g> creates a module-local unique number. It is increased by one (1) with each use. This number is guaranteed to be unique across all clone zones. Defaults to <none> (this means that clone zones uses its default

Name	Description
	naming scheme for groups which is equivalent to `<o>-<uid>`
identical	If this attribute is set to true, each time a clone cycle is run, the cloned units receive identical name and ID as the template. This results in clones that • Replace any previous clones from this template that were cloned with 'identical = true' • Can still be different in location, coalition etc. *identical* and *nameScheme/groupScheme* are mutually exclusive. When identical is true, it overrides any nameScheme settings. Defaults to false
requestable	If set to true, this clone zone can be controlled via other modules, such as HeloTroops. Availability of this cloner to other modules is controlled by those modules (for example, HeloTroops will only control cloners that clone units that can be carried with a helicopter) Note that "requestable:true" and "source:<templates>" are mutually exclusive, so you will se a warning when the cloner module starts up and a zone sports both "requestable" and "source" attributes Defaults to <none>
cooldown	If set to a value greater than zero, a cloner can only start a new clone cycle after <cooldown> seconds have passed since the last clone cycle. Defaults to -1 (no cooldown)
useAI	Only works for ground and naval units. If set to false, the spawned groups have their AI disabled and will neither move nor shoot nor otherwise react to enemies. Defaults to true (AI is turned on)
damaged!	Whenever the current batch of clones receives so much damage that one or more units are destroyed, a signal is sent on the connected flags. Use this when you want to be notified whenever your last clone batch receives significant damage, but that damage is not enough to entirely destroy all units. Note that only AI units are counted, static objects are not monitored. Defaults to <none>
health#	Direct output. Always contains current health in percent of the **current batch of clones** (AI units only, does **not** include static objects). 100 means 100% (all units are still alive), 0 means all units are dead. Defaults to <none>

5.3.12.11 Persistence

Clone Zones support persistence. They persist all groups and static objects that they have spawned (if they are still alive at the point of persistence), and re-connect their spawns to correctly trigger *empty!* if that flag is connected to outputs.

Note that just like groups that you placed with ME, any group that has a route with more than one waypoint (in addition to the initial waypoint) will (due to the way DCS works) immediately

head for the first waypoint. Keep this in mind when adding groups with routes to spawners in missions that you also want to use persistence with.

Persisted Cargo will correctly be managed by cargoManager upon reload.

5.3.12.12 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.

5.3.13 objectSpawnZone

5.3.13.1 Description

ObjectSpawnZones are similar to SpawnZones 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 DCS objects are inanimate, they *can* be linked to other units (ships) and picked up as cargo by helicopters. ObjectSpawnZones have 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 subtle differences to how Object Spawne Zones work versus how Unit/Group SpawnZones do, so make sure that you consult the documentation and understand the various ME Attributes.

Spawn Cycle

Spawning logic per object spawner is as follows:

- When the spawner is monitoring objects (the last-spawn) and all objects from the last spawn have disappeared (by being destroyed, or otherwise removed from the spawner's oversight, e.g. they were transferred to cargoManager or a cargoReceiver), start a cooldown timer
- Determine if spawner should spawn: If any of the following is true
    - The "*spawn?*" input has triggered
    - the cooldown timer has run down and
        - the spawner is not paused
        - the number of performed spawns does not exceed the maximum number of spawns
    - Mission is starting up and spawner is not paused
- If the spawner performs spawning:
    - Assemble a group name based on the *nameBase* attribute. If no nameBase attribute is given, DML automatically generates a safe nameBase for the spawner.
    - Create objects as described in the *types* attribute, one object per type. Repeat this *count* times (so an object spawner with three items in types and count = 4 creates 3x4=12 objects). If no types are given, a single "White_Tyre" type is used. If no count is given 1 is used.
    - For each object, assemble a name based on *nameBase*.
    - If the object spawn zone is linked to a unit (usually a ship), and the spawner's *autoLink* attribute is true, the object is linked to the same unit that the spawner is linked to, using the spawn zones relative offset to the linked unit.

- Arrange all new objects as described in the 'formation' attribute. If no formation is given, the objects are arranged in '*circle_out*'.
- If isCargo is true, and the spawner is managed, the created objects are passed to the cargo manager module (if that module is loaded into the mission) for management
- Place the group inside the spawn zone. If only one object is spawned, it's placed in the middle of the zone.
- Unless the *autoRemove* attribute is set to true, place the spawned group under this spawner's oversight. Any previously monitored groups are forgotten.

Note that unlike unit spawners, object spawners do not check for ownership of the spawner and will happily spawn enemy-owned objects if the spawner was captured.

After objects are spawned, the ObjectSpawnZone keeps an eye on the spawned objects. Once all of them have disappeared from the game (for example 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 cargo simply because a cargo object was picked up. CargoReceivers (see below) have the ability to auto-delete cargo on delivery so this can then trigger the spawner's re-spawn cycle.

When autoRemove is set to true the spawner immediately undergoes a new spawn cycle after spawning (cool-down, then spawn).

Pausing the Spawner
Unless paused, spawn cycles happen automatically at start, and when the spawned objects disappear. You can prevent this by setting the paused attribute (prevents spawning at start), or using the pause? input to deactivate the spawner. When paused, you can only start a new spawn cycle by forcing it via API or spawn? input. You can activate a paused spawner by sending a signal on the activate? input.

Making spawns "delicate" or "brittle"
You can assign a Delicate Zone to the spawner. If you do so, the named Delicate Zone is used to assign delicate status to the spawns, making them explode when they receive a tiny amount of damage. See Delicates module for more info.

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.

Controlling spawn names
You can control how spawned objects are named with the 'baseName' attribute. Unit and object names in DCS missions are special in that at any time there must be at maximum one unit or object with that name. If a spawner or cloner dynamically creates a new object or unit

with a name that already exists in the mission, the existing unit/object is immediately removed.

DML spawners create names for newly spawned units/objects by taking the spawner's 'base name' and adding a spawner-individual count that increases with every unit/object created. Since DML automatically assigns safe base names by default, all spawners create safe unit/object names that cannot come in conflict with each other.

Mission designers can override DML's default behavior and provide their own *baseName* for spawns. In that case, the onus is on the designer to ensure that no name conflicts arise. Since in many cases it's desirable (and safe) to use the spawner's zone name as baseName, DML provides a special convenience shortcut "*" for the name attribute. In that case, the baseName for that spawner is set to the zone's name.

If you need tighter control over a spawned unit/object's name, consider using a clone zone and that zone's "*nameScheme*" or "*identical*" attributes.

Inputs

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 have inputs, and when triggered, the spawner spawns anew *without checking max spawns or cooldown*.

Name	**Description**
f?	Input Watchflag.
activate?	Input Watchflag. Used to 'activate' a paused spawner
pause?	Input Watchflag. 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.

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.

5.3.13.2 Dependencies
Required
ObjectSpawnZones requires dcsCommon, cfxZones

Optional:
cargoManager (for managing cargo events).

5.3.13.3 Module Configuration
ObjectSpawnZones does not require any configuration

5.3.13.4 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., "doSpawn") 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. Defaults to <none>
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 Defaults to <none>
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 spawner Defaults to <none>
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

Name	Description
	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) Defaults to “White_Tyre” **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 (1)
autoTurn	When *count* (see above) is greater than 1, the objects are distributed over the zone’s circumference. If autoTurn is on, all objects are also turned by the same amount that they are rotated around the center. The result is that, e.g. a soldier that faces inward always faces inward. Defaults to false (all objects face the same heading)
country	The country for which the static objects are spawned. Examples: 0 = Russia, 1 = Ukraine, 2 = USA etc. Defaults to 2 (USA)
baseName	A designation (e.g. “Hill Marines”) that is used to name units and groups from during unit spawning. **If provided, you must ensure that resulting unit and group names are UNIQUE.** If you do not assign a base name, a safe baseName that is guaranteed to prevent possible name conflicts is generated. A **convenience shortcut “*”** replaces baseName with the name of the zone. This is a safe baseName and can be used for all spawns. If two spawners have the same baseName, spawned units from one spawner may lead to existing (previously spawned) units being removed from the mission if they have the same name. So if for some reason a spawner appears not spawn units, make it a habit to check for potential name conflicts first. Default: <auto-generated safe baseName>.
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). Set to false if the spawner should ‘drop’ the objects to the ground. Defaults to true (unit is autolinked if spawner is linked)
heading	Orientation (in rads) 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.

Name	Description
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 spawnObjects?). 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
useDelicates	Name of a Delicates Zone that is used to assign *delicate* (brittle, explodes when receiving minimal damage) status when this spawner spawns objects You can use an asterisk (“*”) as wildcard to refer to this zone Defaults to <none>

5.3.13.5 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

5.3.14 cargoReceiverZone

5.3.14.1 Description

This module solves a Mission Editor limitation: unlike ME plain vanilla triggerZones, DML's cargoReceiverZones *can* generate signals when players unhook cargo inside. 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 spawn the cargo objects) and cfxCargoManager module that tracks the cargo objects and provides the required cargo events.

Inputs and Outputs

Cargo receiver zones can send signals on their outputs, allowing mission designers to trigger on a delivery. This is controlled by adding attributes to the zone:

Name	Value	Description
cargoReceiver!		Output flag to signal when the cargo object is delivered onto the ground inside the zone.
~~f! cargoReceived!~~	~~Number~~	~~Sets this flag according to method each time cargo is delivered into the zone~~

5.3.14.2 Dependencies

CargoDeliveryZones can only track cargo that is registered with cargoManager

It therefore requires that the following modules have loaded:

dcsCommon, cfxZones, cargoManager

You usually also want to add *objectSpawnZones* to your mission because they create cargo objects that players can pick up and deliver

5.3.14.3 Module Configuration

(none)

5.3.14.4 ME Attributes

Name	Description
cargoReceiver!	Output flag to bang! when the cargo object is delivered onto the ground inside the zone. Marks this zone as a cargo receiver zone. **MANDATORY**
~~cargoReceiver~~ **deprecated**	~~Marks this zone as a cargo receiver zone. Value is ignored~~ **deprecated**
autoRemove	Delete (despawn and remove from game) a cargo object <removeDelay> seconds after it was successfully delivered. This is helpful for most ObjectSpawnZones set-ups to trigger their spawn cycle Defaults to false

Name	Description
silent	Set to true to turn off this zone's directions. Defaults to false (zone will talk to pilots)
method cargoMethod	DML Method for output Defaults to "inc"
~~f!~~ ~~cargoReceived!~~ **deprecated**	~~The flag to bang! when the object is destroyed. Use only one synonym per zone.~~
removeDelay	The delay (in seconds) after which after a successful delivery of a cargo object is removed. Requires that autoRemove be set to true. Defaults to 1 (second), Minimum is 1
noBounce	Prevents multiple deliveries of the same cargo object to the same zone are counted multiple times (e.g. bounces). When set to true, each cargo can only be delivered once per receiver zone. Defaults to true (one delivery per zone)

5.3.14.5 Using the module

Add the cargoReceiverZone source into a DOSCRIPT action at the start of the mission.

Place a cargo receiver zone.

5.3.15 artilleryZones

5.3.15.1 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 use 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.

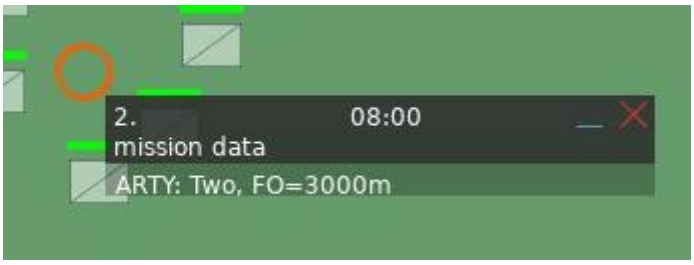
ME Flag Integration

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)

5.3.15.2 Dependencies

cfxArtilleryZones requires dcsCommon and cfxZones

5.3.15.3 Module Configuration
None.

5.3.15.4 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
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

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

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

Place artillery zones with ME

5.3.16 Artillery UI

5.3.16.1 Description

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

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

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

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

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

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

Target Direction / Guidance

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



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

When a target is reported as OBSERVING

Marking Zones

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

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

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

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

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

5.3.16.3 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

5.3.16.4 ME Attributes
None.

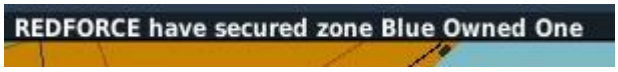
5.3.16.5 Using the module
Include the cfxArtilleryUI source into a DOSCRIPT Action at the start of the mission

Optionally, add a config zone with ME

5.3.17 ownedZones

All zones in DML implicitly "belong" to a faction. By default, that faction is "Neutral", and by explicitly adding an "owner" attribute to a zone, you can change that zone's ownership to another function (red or blue). Once you have assigned ownership of a zone, that ownership remains static throughout the entire mission. Unless you add the "Owned Zones" module to your mission: **the "OwnedZones" module makes zones conquerable** by other factions.

5.3.17.1 Description

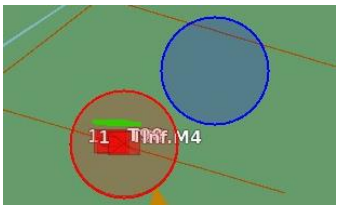


Owned Zones is a module that manages 'conquerable' zones and keeps a record of which coalition owns which zone.
Ownership is updated regularly. Owned Zones anchors itself to zones with the 'owner' attribute set from within ME.

Note

'owner' is an attribute that *all* DML Zones share: it is assigned implicitly by DML and is set to neutral by default. You can set it **explicitly** by adding an 'owner' attribute in ME ***and including this module.*** As soon as you do that, that zone's ownership becomes dynamic.

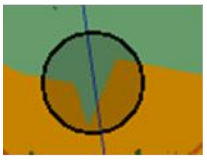
Visuals – Color Management



Owned zones are shown on the F10 Map in-game and by default 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.

You can change the default colors for red, blue and neutral globally in the config zone, and additionally change the colors of individual zones in the zone attributes using the *xxxLine* and *xxxFill* attributes. When you supply your own colors, they are in RGBA format, meaning that you give values (all ranging from 0.0 to 1.0) for each channel Red, Green, Blue and Alpha (=opacity). For example, *yellow, full opacity* would be "`1.0, 1.0, 0.0, 1.0`", while *blue, half opacity* would be "`0.0, 0.0, 1.0, 0.5`".

Name	Value
owner	neutral
neutralLine	0.0, 0.0, 0.0, 1.0
neutralFill	0.0, 0.0, 0.0, 0.2
blueLine	1.0, 1.0, 0.0, 1.0
blueFill	1.0, 1.0, 1.0, 0.2



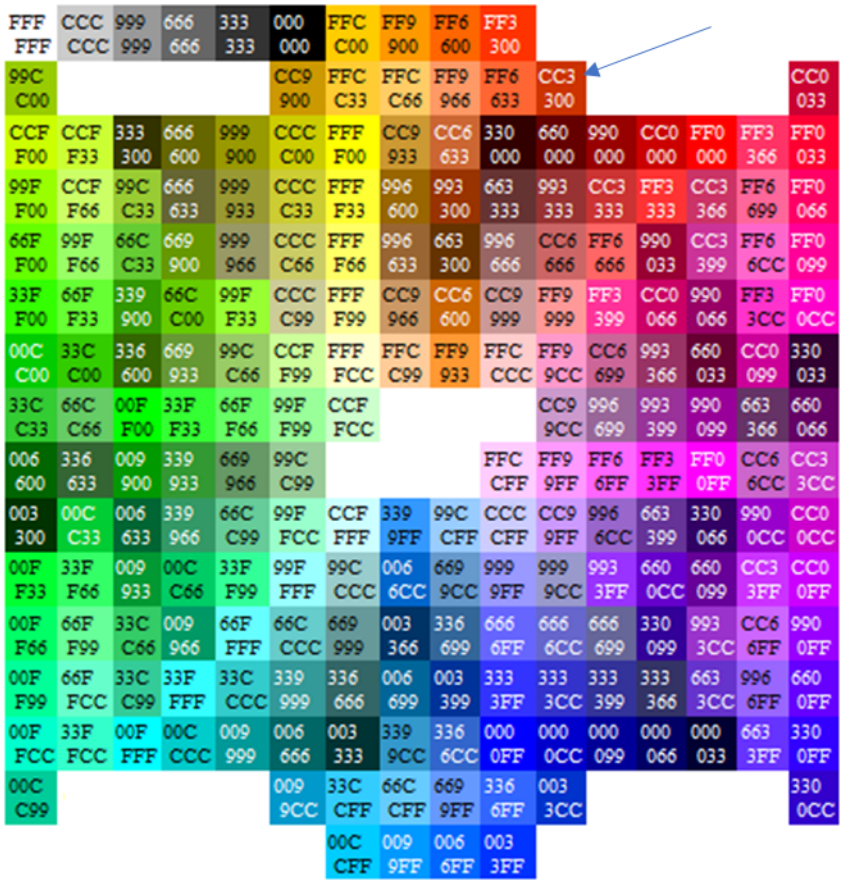


In the example to the right, we have defined the zone to show up as black when neutral, and a yellow outline with 20% opaque white fill when belonging to blue. Owned zones inherit the color from the config zone if you don't supply your own colors, and if you omit colors from the config zone altogether, they revert to their standard red, blue and grey.

Alternatively, you can enter colors as ‘hex strings’ as you would in some web editors / CSS. The format here is “#RRGGBBAA”, with a leading hash mark, and then two hexadecimal digits per color:

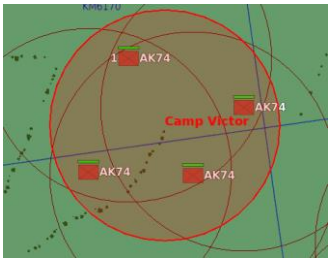
For example, the red color swatch indicated by the blue arrow on the right has the color code “CC3300”. If you enter sich a color, DML will substitute an alpha value of FF (100%) automatically.

If you are looking for more colors and their values, search the web for “RGBA color values”.



Zone Labels/Titles

Optionally, owned Zones can also display some text on the F-10 map. You control what text is displayed with the ‘title’ attribute. If you omit the attribute, no title is displayed. The title’s text is always displayed in the color that you have defined as line color for the owning faction, and its first character is placed at the zone’s center. If the zone changes hands, so will the color of the text (to match that of the new owner)



Interaction with other Modules that can manage ‘owner’

Obviously, in a military game like DCS, owning and – more importantly – capturing zones can become a major part of a mission. DCS itself knowns the concept of ownership for some of its game elements, and DML integrates this with additional modules, who then build on this concept and can extend it:

ownedZones

This is DML’s bedrock module to provide generic zones with the ability to be owned, and detects if one side conquers such a zone (i.e. takes it from the other side or NEUTRAL). To add this ability to your mission, you must include the ownedZones module. Owned zones can inherit their ownership from other zones with the *masterOwner* attribute.

FARPZones

FARPs are special static objects in DCS that you can place with Mission Editor. Their ownership is managed by DCS, not DML. DML’s FARPZones module allows a DML-style

interface (sending signals when ownership changes), and owned zones can inherit ownership from FARPZones

Airfields
Much like FARPS (which are little more than user-placable helicopter airfields), airfields have their ownership managed by DCS. Unlike FARPS, mission designers cannot place airfields on a map. DML's airfield module provides access to airfield ownership similar to the FARPZones module. Owned zones can inherit ownership from airfield zones.

ownAll
Since ownership of zones can be important from a strategic aspect, DML provides a module that can provide a meta- or hierarchical aspect to ownership: ownAll zones derive their own ownership from the owner of the zones that they link to.

Conquering an Owned Zone
An Owned zone is conquered when there are only troops belonging to one (conquering) faction inside the zone left alive, and this number is equal to, or exceeds, the number of units required to capture the zone (this can be configured with the *numCap* attribute).

A single unit (including infantry parachuted in) can therefore conquer a zone, as long as there are no units from the opposing faction inside the zone, and this fulfils the minimum unit constraint (default is 1 unit). Capturing a zone is instantaneous once it fulfills capture conditions. 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.

When a zone changes owners, it can create several flag events/signals that you can use in your mission to trigger other actions.

Units that can capture an Owned Zone
By default, only ground units can capture an owned zone – zooming with a jet through an owned zone will not do it. You can, however, use ownedZone's *xxxCap* options to change this:

- *groundCap*
  Allows ground units to capture an owned zone. Enabled by default.

- *navalCap*
  Allows ships to capture an owned zone. This of course pre-supposes that the owned zone is somewhere where ships can go (and ground units usually can't). Disabled by default

- *heloCap*
  Allows helicopters to capture an owned zone. This requires that the helicopter has landed inside the zone, an airborne helicopter cannot capture an owned zone. Disabled by default

- *fixWingCap*
  Allows fixed-wing aircraft to capture an owned zone. This requires that the fixed-wing aircraft has landed inside the owned zone. Disabled by default.

Note that enabling more than one type of units that can capture owned zones can negatively impact performance when there are a lot of units deployed.

Keeping an Owned Zone
By default, a faction keeps a zone that they conquered until it is conquered by the opposing faction. However, mission designers can require that the holding faction keep a minimum amount of ground units inside the zone lest it loses the zone (in that case the zone becomes NEUTRAL, and requires capture again). You can control this with the *numKeep* attribute in the config zone (which defaults to 0).

Contested Zone Status / Easy Contest
When a faction holds a zone, certain actions from the opposing side can cause a zone to become 'contested' – in this case the zone becomes NEUTRAL, and requires that one side fulfils the capture requirements to gain control.

OwnedZones provides a special 'easyContest' config option that makes any zone contested that contains ground units from both factions (red and blue). This means that if red holds a zone, a single blue unit inside the zone will turn it neutral (contested). By default, easyContest is turned off.

Zone Protection Attribute unbeatable/untargetable
There are several attributes that can protect an owned zone from being captured by 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=true" so they are never conquered by another faction.
- Owned Zones can be set to "untargetable=true" so that DML AI will ignore them when looking for a zone to attack. Note that DCS AI will still attack units in this zone if they encounter each other.

Zone Protection Attribute untargetable
You can add an attributes that prevents produced units to attack a factoryZone. You can use this to prevent certain conditions from arising (such as a critical factory zone is inadvertently taken out by AI instead of players).

- factoryZones can be set to "untargetable=true" so that DML AI will ignore them when looking for a zone to attack. Note that DCS AI will still attack units in this zone if they encounter each other.

Keeping Track Of Zones
Owned zones provides a host of output flags that you can use to track when individual zones change hands, as well as higher-level outputs to get the number of zones that a faction

owns. There are also two one-shot flags (i.e. outputs that will at most trigger once per mission) that can tell you when BLUE or RED own all ownedZones on the map:

- Per Zone Outputs (fire when that zone changes hands)
  - Conquered!
  - redcap!
  - redLost! (also fires when zone becomes contested)
  - blueCap!
  - blueLost! (also fires when zone becomes contested)
  - neutral! (also fires when zone becomes contested)
  - ownedBy# (always carries the faction 0/1/2 that currently owns the zone)

- Map-Wide Outputs (defined in ownedZonesConfig)
  - r# (increases when red captures a zone, decreases when red loses a zone. Can become negative)
  - b# (increases when blue captures a zone, decreases when blue loses a zone. Can become negative)
  - n# (increases when a zone becomes neutral, decreases if a neutral zone becomes red or blue)
  - redOwned# - number of zones that RED currently owns
  - blueOwned# - number of zones that BLUE currently owns
  - neutralOwned# - number of zones that NEUTRAL currently owns
  - totalZones# - number of ownedZones in the mission. This number currently does not change while the mission is running.

- One-Shot Own All Outputs (defined in ownedZonesConfig)
  - allBlue! fires once in the event that BLUE owns all zones on the map
  - allRed! fires once in the event that RED owns all zones on the map

Inheriting Ownership
Owned zones can 'inherit' ownership from other zones by use of the masterOwner attribute. In this case, a zone's ownership isn't determined by the number/faction of the units inside the zone, but **solely by the owner of the zone that is the masterOwner**.

Note that zone ownership can lag a cycle or more if you cascade zone ownership over multiple instances: If ownership of zone C is determined by B via masterOwner, and B's ownership is determined by A, it can take up to two cycles for a new ownership to 'trickle down' from A→B→C. A 'cycle' is the time interval between ownedZones' checks for zone ownership, and is set to 1 second by default.

ME Flag Integration
Owned Zones supports live count outputs for when red, blue and neutral gain or lose control of zones (map-wide). Owned Zones changes the relevant outputs each time it changes hands. The logic is that the output flag's value 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 *output!* (bang!) flags are **set in the zone's configuration zone**, as they apply to all owned zones.

Name	Description
n#	Increase this flag by one if neutral wins an owned zone or it becomes contested (in which case it turns neutral). 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#	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#	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
redOwned#	This flag holds the current running total of zones owned by red
blueOwned#	This flag holds the current running total of zones owned by blue
neutralOwned#	This flag holds the current running total of zones owned by neutral
totalZones#	This flag holds the total numbers of owned zones in the mission. Currently does not change during mission (is constant), this may change in the future
allRed!	The first time that RED owns all ownedZones on the map, value of this flag increases by one. Can only happen once
allBlue!	The first time that BLUE owns all ownedZones on the map, value of this flag increases by one. Can only happen once

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

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, the owning faction defaults to neutral (0) **MANDATORY** **Note:** "owner" is a zone attribute that is implicitly available for **all** zones in DML. By default, all zones are owned by the neutral faction. All zones support the 'owner' attribute, as it is a core zone ability. By adding the "OwnedZones" module to your mission, you merely change the static ownership of a zone to a dynamic property managed by the OwnedZones module. **Note II:** Originally (pre version 2 of ownedZones), this module combined the functionality of ownedZones and factoryZones. All production ability has been moved to factoryZone.
conquered!	Flag to bang! when this zone changes hands Defaults to <none>
redCap!	Flag to bang! when red captures this zone Defaults to <none>
redLost!	Flag to bang! when red loses control over this zone. This includes the zone turning neutral/becoming contested Defaults to <none>
blueCap!	Flag to bang! when blue captures this zone Defaults to <none>
blueLost!	Flag to bang! when blue loses control over this zone. This includes the zone turning neutral/becoming contested Defaults to <none>

Name	Description
neutral!	Flag to bang! when this zone becomes neutral or contested Defaults to <none>
ownedBy#	Flag that is always set to the number of the owning faction: 0 (neutral), 1 (red) or 2 (blue)

5.3.17.2 Dependencies
Required: dcsCommon, cfxZones

5.3.17.3 Module Configuration
To configure the Owned Zones module via a configuration zone,

- Place a Trigger Zone anywhere on the map
- 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 false
announce announcer	Show a message and play a sound if an owned zone is captured. Default is false
n#	Increase this flag by one if neutral wins an owned zone. Decrease this flag by one if neutral loses an owned zone Defaults to <none>
r#	Increase this flag by one if red wins an owned zone. Decrease this flag by one if red loses an owned zone Defaults to <none>
b#	Increase this flag by one if blue wins an owned zone. Decrease this flag by one if blue loses an owned zone Defaults to <none>
redOwned#	This flag holds the current running total of zones owned by red Defaults to <none>
blueOwned#	This flag holds the current running total of zones owned by blue Defaults to <none>
neutralOwned#	This flag holds the current running total of zones owned by neutral Defaults to <none>
totalZones#	This flag holds the total numbers of owned zones in the mission. Currently does not change during mission (is constant), this may change in the future Defaults to <none>
allRed!	The first time that RED owns all ownedZones on the map, value of this flag increases by one. Can only happen once Defaults to <none>
allBlue!	The first time that BLUE owns all ownedZones on the map, value of this flag increases by one. Can only happen once Defaults to <none>
numCap	Minimum number of ground units required to capture this zone. No units of the opposing faction must be inside the zone, neutral units are not counted. Defaults to 1 (a single unit can capture a zone)

numKeep	Minimum number of ground units required inside the zone to keep a zone *after* capture. If less units than this number are inside the zone, it becomes neutral. Note that the mere presence of enemy units inside an owned zone usually (see exception below) does not make it contested/neutral. Defaults to 0 (no units required to keep a zone after capture)
easyContest	Usually, a zone that is owned by red or blue becomes contested only when the number of ground units from the owning side falls below numKeep. If you set easyContest to true, any zone that contains ground units from both factions (red and blue) becomes neutral/contested. That way, the presence of a single blue unit inside a heavily fortified red zone turns it neutral, and remains neutral until the capture constraints are met by one side. Defaults to false
winSound	Name of the sound file to be played to winning faction whenever the opposing side loses a zone This sound is also played when a zone formerly owned by the opposing faction turns neutral.
loseSound	Name of the sound file to play to the faction that is losing a zone
groundCap	When set to true allows ground units to capture an owned zone. Defaults to true
navalCap	When set to true allows ships to capture an owned zone. Owned zone must be inside water Defaults to false
heloCap	When set to true allows helicopters to capture an owned zone. This requires that the helicopter has landed inside the zone, an airborne helicopter cannot capture an owned zone. Defaults to false
fixWingCap	When set to true allows fixed-wing aircraft to capture an owned zone. This requires that the fixed-wing aircraft has landed inside the owned zone. Defaults to false
redLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone's outline when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimal numbers (as used in HTML/CSS to denote colors) Defaults to "1.0, 0, 0, 1.0" or "#FF0000FF", a deep red, fully opaque
redFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimal numbers (as used in HTML/CSS to denote colors) Defaults to "1.0, 0, 0, 0.2" or "#FF000033" an 80% transparent red
blueLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone's outline when it belongs to the BLUE faction.

	RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimal numbers (as used in HTML/CSS to denote colors) Defaults to “0, 0, 1.0, 1.0” or “#0000FFFF” a deep blue, fully opaque
blueFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the BLUE faction. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimal numbers (as used in HTML/CSS to denote colors) Defaults to “0, 0, 1.0, 0.2” or #0000FF33”, an 80% transparent deep blue
neutralLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone’s outline when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimal numbers (as used in HTML/CSS to denote colors) Defaults to “0.8, 0.8, 0.8, 1.0” or “#CCCCCCFF” a light gray, fully opaque
neutralFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimal numbers (as used in HTML/CSS to denote colors) Defaults to “0.8, 0.8, 0.8, 0.2” or “#CCCCCC33”, an 80% transparent grey
method	DML method for output signals that aren’t direct (“#”) outputs Defaults to “inc”
excludedTypes	Sometimes, you may find it helpful if you can exclude some types of units to capture a zone. You can add the types that you want ownedZones to ignore as a comma-separated list here. Note that using this option can impact performance, and only works with DML owned zones – those typed will still be able to conquer DCS-controlled capture rules: airfields and FARPS. Example: `Soldier M4 GRG` Defaults to <none>, option is turned off

5.3.17.4 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** **Note:** “owner” is a zone attribute that is implicitly available for **all** zones in DML. By default, all zones are owned by the neutral faction. All zones support the ‘owner’ attribute, as it is a core zone ability. By adding the “OwnedZones” module to your mission, you merely change the static ownership of a zone to a dynamic property managed by the OwnedZones module. **Note II:** Originally (pre version 2.x of ownedZones), this module combined the functionality of ownedZones and factoryZones. All production ability has been moved to factoryZone.
conquered!	Flag to bang! when this zone changes hands Defaults to <none>
redCap!	Flag to bang! when red captures this zone Defaults to <none>
redLost!	Flag to bang! when red loses control over this zone. This includes the zone turning neutral/becoming contested Defaults to <none>
blueCap!	Flag to bang! when blue captures this zone Defaults to <none>
blueLost!	Flag to bang! when blue loses control over this zone. This includes the zone turning neutral/becoming contested Defaults to <none>
neutral!	Flag to bang! when this zone becomes neutral or contested Defaults to <none>
ownedBy#	When present, the flag specified here is always set to the currently owning faction: 0 (neutral), 1 (red) or 2 (blue). This can be used to create win conditions, and changes on ownership will trigger standard ‘change’ inputs Defaults to <none>
unbeatable	“true” or “yes” makes it unbeatable/unconquerable. This zone can’t be conquered by the other side, but may still be targeted. Defaults to “no”
untargetable	“true” or “yes” makes it untargetable. Zone will not be targeted by troops with ‘attackOwnedZones’. The zone may still be conquered by the other side. Defaults to “no”
hidden	“true” or “yes” hides it. Zone is not shown on F10 Map. Defaults to “no”
title	If present, the value (text) is drawn starting at the center of the zone, in the lineColor of the current owner. The special wildcard “*” can be used to denote that the title is the same as the zone’s name. Defaults to <none> - no title is drawn.
redLine	Four numeric values, separated by comma (e.g. 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the

Name	Description
	color used to draw the zone's outline when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Defaults to config zone's setting of *redline*
redFill	Four numeric values, separated by comma (e.g. 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Defaults to config zone's setting of *redFill*
blueLine	Four numeric values, separated by comma (e.g. 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone's outline when it belongs to the BLUE faction. RGBA values each range from 0.0 to 1.0 Defaults to config zone's setting of *blueLine*
blueFill	Four numeric values, separated by comma (e.g. 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the BLUE faction. RGBA values each range from 0.0 to 1.0 Defaults to config zone's setting of *blueFill*
neutralLine	Four numeric values, separated by comma (e.g. 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone's outline when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Defaults to config zone's setting of *neutralLine*
neutralFill	Four numeric values, separated by comma (e.g. 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Defaults to config zone's setting of *neutralFill*
method	DML output method for all outputs ("!") that aren't direct ("#") Defaults to "inc"
numCap	Local override for the global numCap setting in ownedZonesConfig. Minimum number of ground units required to capture this zone. No units of the opposing faction must be inside the zone, neutral units are not counted. Defaults to ownedZonesConfig's settings
numKeep	Local override for the global numKeep setting in ownedZonesConfig. Minimum number of ground units required inside the zone to keep a zone *after* capture. If less units than this number are inside the zone, it becomes neutral. Note that the mere presence of enemy units inside an owned zone usually (see exception below) does not make it contested/neutral. Defaults to ownedZonesConfig's settings
announcer announce	Announce capture by text and sound if this zone is captured. Overrides and defaults to config's *announcer* attribute setting.

5.3.17.5 Persistence

OwnedZones supports persistence.

5.3.17.6 Using the Module

To enable, add the script to the mission as a DOSCRIPT action during Mission Start. To configure the module, place a 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.

5.3.18 factoryZones

This module adds the ability to 'produce' ground units in the form of defenders and attackers to a zone. factoryZones work closely with ownedZones (earlier DML versions of factoryZones were integrated into ownedZones) and have abilities to react to change of ownership of the zone.

Although factoryZones can produce units (based on "type strings" that you provide), they now also provide outputs! that you can use to connect spawners and cloners to, and factory zones then provide the signals that you need to trigger those modules.

Note:

Prior to version 2.x of factoryZones and ownedZones, the functionality of factoryZones was contained within ownedZones. The separated factoryZones are backward-compatible (on attribute level) with ownedZones version 1.x.

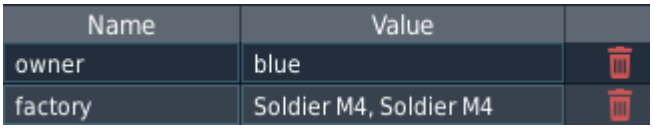
5.3.18.1 Description

factoryZones produce ground units to defend the zone, and - once defenders are produced - the factory then can produce units to attack enemy-owned zones.

Controlling Production of Units / Ownership Options

A factoryZone produces units with different purposes (orders – see DML's section on Orders):

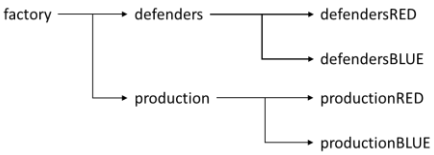
- defenders to defend the factory (produced first). These units are issued '`guard`' orders
- offensive troops to seek out and attack enemy-owned zones. These units are issued '`attackOwnedZone`' orders.



You have great flexibility to choose the type and number of units that a factory produces to defend itself and attack other zones with. On the most basic level, you use the "*factory*" attribute to tell a factory what to produce. It then creates these units both as defensive and attacking units. If, for example, your *factory* attribute is set to "`Soldier M4, Soldier M4`", the factory will first produce a group consisting of two "Soldier M4"-type units as defenders that have orders to guard the zone, and that belong to the faction that currently owns the zone. Once that is complete, it starts producing the same group consisting of two "Soldier M4"-type units as attackers with orders to attack the nearest enemy owned zone.

The unit types that you list as value for the various unit attributes are standard DCS unit types as defined in the typeName attribute of each unit's description table in DCS's master unit DB (documented here: https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB)

You can refine unit production by providing a '*defenders'* or '*production'* attribute. In that case the factory can produce different types of units for defensive and offensive units.



To support dissimilar or asymmetric scenarios, factories also can further produce different units depending on the faction that currently owns the factory. Use the *productionRED*, *productionBLUE* attributes to specify the type and number of offensive units (units that seek out enemy owned zones) and *defendersRED* and *defendersBLUE* to do the same for defending units per side. To make factory production as flexible and usable as possible, the attributes that control what is produced follow a simple "type inheritance" tree. To determine what unit types are produced, the 'furthest' defined leaf in the tree is used.

Example:
Let's assume that '*defenders'* is the only type attribute that we define in a factory zone, and it is set to '`BTR-80, Soldier M4`'. This means that it produces the following units:

Name	Value	
owner	blue	
factory		
defenders	BTR-80,Soldier M4	

- Defensive units: *defendersRED* and *defendersBLUE* both inherit from the *defenders* attribute, hence both factions produce the same two ground units: one BTR-80 and one M4 Infantry Soldier.
- The factory produces *no* attacking units: *productionRED* and *productionBLUE* inherit from the *production* attribute, which inherits from the factory attribute which – since its value field is empty – defaults to the 'none' meta-type, meaning that no ground units are produced.

Using Cloners (cloneZones) and Spawners for Production
Instead of / in addition to using type strings (as described above) you can use a factory's attributes *redP!*, *redD!*, *blueP!* and *blueD!* to trigger cloners (or other spawners) to produce units according to their templates. These attributes follow the following logic:

- redX! triggers if a RED production is cycled, and
- blueX! will likewise only trigger if BLUE produces.
- xxxP! is triggered for a standard production cycle,
- xxxD! Is triggered when defenders are to be produced.

What you connect to those outputs is your decision. Remember that when you are using cloners/spawner to produce defenders, the factory will *not* track their status and therefore also will not repair them should they become damaged; you will have to configure the cloner / spawner for such a purpose.

Note also that only a factory or spawner can produce troops that automatically seek out other owned zones; clone zones create a copy of their template, and every template already has orders.

Factory Zones MUST Also Be OwnedZones
It is imperative that you also supply an 'owner' attribute to each factory zone, which – since factoryZones requires the ownedZone module – automatically also turns a factory

zone into an owned zone. If you need a factory to always remain in the hands of one faction, use the '*unbeatable*' and '*untargetable*' attributes from ownedZones to protect the factory's ownership.

Since factoryZones requires OwnedZones as a dependency, all factoryZones can potentially change hands (unless prevented from doing so by using ownedZone's "*unbeatable*" attribute (this requires that the factory zone is also an owned zone which is indicated by having an 'owner' attribute).

If you omit the `owner` attribute for a factory zone, it's ownership defaults to neutral. Neutral factories do not produce units, so if a factory seemingly is on strike, make sure that it's ownership isn't neutral (in other words: make sure that it is either RED or BLUE)

Defenders / Attackers Production Logic

factoryZones spawn troops to defend the zone (defenders), and they can send spawn troops (for example to actively engage **owned zones)**. What troops a factory produce are determined with the 'defendersRED/BLUE' and 'attackersRED/BLUE' attributes, in conjunction with the currently owning faction of that zone. Neutral zones do not produce attackers nor defenders.

The logic for production is as follows:

- When the mission starts up, defenders for the faction that owns the zone are produced instantly. If the zone is neutral, no defenders are created.
- When a factory zone is captured (note: **all factory zones must also be ownedZones**), the zone enters a "defender production" cycle (wait cycle). If at the end of the wait cycle the zone is still held by the same faction,
    - the xxxD! flag is banged! if defined (redD! when the zone is held by RED, blueD! when the zone is held by BLUE)
    - defenders are spawned. What defenders are spawned is determined by the production and defender attributes.
- 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 ownedZones to attack, the zone spawns no units.**
    - the xxxP! flag is banged! if defined (redP! when the zone is held by RED, blueP! when the zone is held by BLUE)
    - A new attacker group (consisting of units as described in attackersXXX) is spawned that automatically seeks out other ownedZones [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.

Once attackers are produced, the module hands them off to cfxGroundTroops with orders to "`attackOwnedZone`".

Livery Support
Starting with version 3.0, FactoryZones support 'liveries' (or paint schemes as they are also called in Mission Editor). You can use this to replace the default texture ('livery', 'skin' or 'paint scheme' for units that are produced by the factory with the one that you specify in the 'factoryLiveries' zone. Note that this change applies globally to all factories.

5.3.18.2 Legacy compatibility
FactoryZones were a part of OwnedZones prior to version 2.0 of owned Zones. The modules retain compatibility with 1.x, and you can upgrade your mission to version 2 of owned zones by following these steps:

- Replace OwnedZones 1.x with the newest OwnedZones version (2.x or later)
- Add FactoryZones to your mission after OwnedZones
- If you have placed an *ownedZonesConfig*, create a copy and name it *factoryZonesConfig*. This will ensure that factory zones production behaves like pre-2.0 OwnedZones production.
- Add a 'factory' attribute to any OwnedZone that is supposed to produce units, and leave the value field empty

5.3.18.3 Dependencies
Required: dcsCommon, cfxZones, cfxOwnedZones, commander, groundTroops

5.3.18.4 Module Configuration
To configure the Owned Zones module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off
defendingTime	Time (in seconds) for producing defenders. Defaults to 100
attackingTime productionTime	Time (in seconds) for factory production. Defaults to 300 (5 Minutes)
shockTime	Time (in seconds) after an attack on defenders before repair commences. Defaults to 200 (some 2 minutes)
repairTime	Time (in seconds) for repairs to complete. Defaults to 200 (some 2 minutes)

Livery Support
You can change the liveries/paint schemes that are applied to units that are produced by the factories on the map. To do so,

- Place a Trigger Zone on the map
- Rename it to “factoryLiveries” (note: exact match required)

Then add name/value pairs for the unit type name (e.g. “Soldier M4” and “winter”) for all the types that you want to replace the default DCS livery with another. If there is no livery of that name available for that unit type, it is spawned with default livery.

Name	Description
Type name for unit, e.g. `Soldier M4`	Name of the paint scheme/Livery as defined by DCS, for example `winter` You must know the correct name, spelling and availability of a livery/paint scheme for a type in order to use this feature. Discovery of these “livery_ID” for any given type is definitely **not trivial** and requires a deep understanding of things DCS; there unfortunately currently is no comprehensive Livery Reference, and the names for paint schemes that are displayed in Mission Editor can be quite misleading.

5.3.18.5 ME Attributes

Name	Description
factory	Enables production abilities and attributes for this zone. The value is a string, coma separated, that tells the factory what types of troops to spawn. Can be overridden by other attributes like ‘*production*’, ‘*defenders*, ‘*productionXXX’* or ‘*defendersXXX’* Example: "`Soldier M4,Soldier M4`” produces two Infantry soldiers for defenders and attackers. **Warning:** these types need to *exactly* match DCS’s types. Be sure not to accidentally insert blanks. Special meta-type: “none” – no troops Defaults to “none” (no troops produced) Note that you **must** also supply a separate ‘owner’ attribute (which will also turn this zone into an owned zone); otherwise, this factory belongs to the neutral faction and can’t be captured **MANDATORY** **Note:** In pre-2.x versions of ownedZones/factoryZones, the ‘factory’ production ability was integrated into ownedZones. If you are upgrading a mission with pre 2.0 version ownedZones to 2.x or later, you gain 99% backward-compatibility simply by adding this ‘factory’ attribute to the owned zone and leaving all other attributes as they are.
owner	**MANDATORY** each Factory must also be an owned zone, the owner attribute

Name	Description
	must be managed by the ownedZones module For value description, please see “ownedZones”
production	A string, coma separated, that tells the factory what types of troops to spawn. Can be overridden by other attributes like ‘*defenders*, ‘*productionXXX’* or ‘*defendersXXX’* Example: "`Soldier M4,Soldier M4`” produces two Infantry soldiers for attackers. **Warning:** these types need to *exactly* match DCS’s types. Be sure not to accidentally insert blanks. Special meta-type: “none” – no troops. Use this if you, for example, intend to use cloners to spawn troops. Defaults to current value of ‘*factory*’ attribute
defenders	A string, coma separated, that tells the factory what types of defensive ground units to spawn. Can be overridden by the side-specific attributes ‘defendersRED’ or ‘defendersBLUE’ Example: "`Soldier M4,Soldier M4`” places two red infantry soldiers to defend the factory. Defaults to current value of ‘*factory*’ attribute
defendersRED	A string, coma separated, that specifies the types of defensive troops to spawn when the zone is owned by RED. Providing this attribute overrides the settings that you may have supplied in ‘*defenders’* or ‘*production’* Example: "`Soldier M4,Soldier M4`” places two red infantry soldiers to defend the factory when it is owned by the red faction. Defaults to current value of ‘*defenders’* attribute
defendersBLUE	A string, coma separated, that specifies the types of defensive troops to spawn when the zone is owned by BLUE. Providing this attribute overrides the settings that you may have supplied in ‘*defenders’* or ‘*production’* Example: "`Soldier M4,Soldier M4`” places two blue infantry soldiers to defend the factory when it is owned by the red faction. Defaults to current value of ‘*defenders’* attribute
productionRED	A string, coma separated, that specifies the types of offensive troops to spawn when the zone is owned by RED. Offensive troops seek out enemy owned zones. Providing this attribute overrides the settings that you may have supplied in ‘*production’* Example: "`Soldier M4,Soldier M4`” places two blue infantry soldiers to defend the factory when it is owned by the red faction. Defaults to current value of ‘*production’* attribute
productionBLUE	A string, coma separated, that specifies the types of offensive troops to spawn when the zone is owned by BLUE. Offensive troops seek out enemy owned zones. Providing this attribute

Name	Description
	overrides the settings that you may have supplied in '*production'* Example: "`Soldier M4,Soldier M4`" places two blue infantry soldiers to defend the factory when it is owned by the red faction. Defaults to current value of '*production'* attribute
redP!	Output to bang! whenever production cycles for "Production" units for the RED faction. Defaults to <none>
redD!	Output to bang! whenever production cycles for "Defender" units for the RED faction. Defaults to <none>
blueP!	Output to bang! whenever production cycles for "Production" units for the BLUE faction. Defaults to <none>
blueD!	Output to bang! whenever production cycles for "Defender" units for the BLUE faction. 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 the zone's production. "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 not unpause the zone. Since paused zones will signal their capture normally, so you can wire the "conquered!" output into "activate?" to automatically activate paused zones upon capture. Defaults to "no"
pause?	DML Watchflag to set paused status! Triggers on change Defaults to <none>
activate?	DML Watchflag to un-pause a paused owned zone. Defaults to <none>
helpMe?	DML Watchflag that triggers a 'defend' cycle (will produce new defenders). Defaults to <none>
triggerMethod factoryTriggerMethod	DML method to trigger inputs Defaults to "change"
method factoryMethod	DML method to bang! on outputs Defaults to "inc"

5.3.18.6 Persistence

A factoryZone keeps track of all units it has produced (attackers and defenders) and will, upon load re-create those groups where they were when the mission was saved. **If these groups are controlled by cfxGroundTroups (as per default), they will resume their tasks immediately** (in sharp contrast to units placed with ME who will tend to proceed to their first waypoint). This currently only applies to troops with orders ‘attack zone’, ‘lase’, and ‘attackOwnedZone’.

5.3.18.7 Using the Module

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

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

Then, place Trigger Zones in ME. Add the ‘factory’ property and add other properties as required.

5.3.19 FARP Zones

5.3.19.1 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.

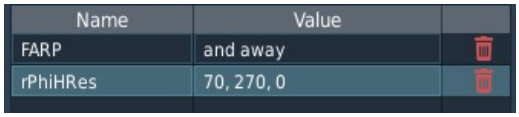
When creating a FARP zone, it is best to place it on, or very close to, the center of the FARP static object itself, so that the Resource Vehicles are easy to place, and you can ensure that the FARP is contained within the Zone.

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 captured, though, the FARP can behave unpredictably, as it is not defined which FARP Zone will receive the notification.

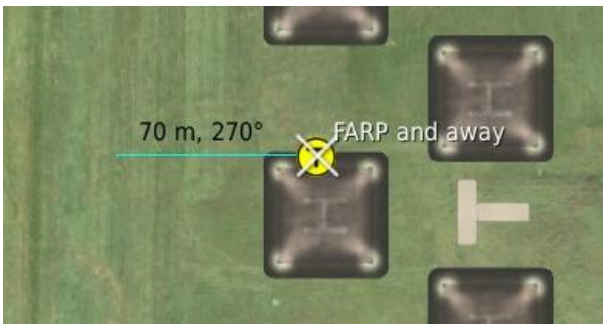
FARPs are marked by a circle in the F10 player map, colored in the color of the owning faction.

Note:
FARP Zones do not work with Airfields.

r-Phi-What???



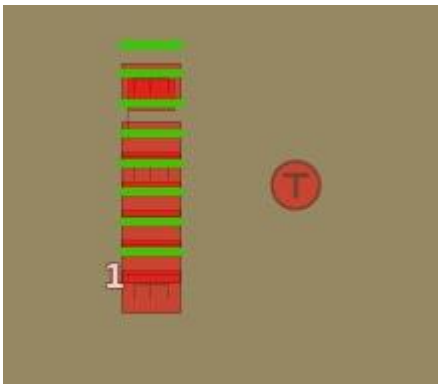
In order to place the resource and defensive units, FARP uses a system called ‘polar notation’. It’s very easy to use in ME, and you’ll soon find out why. Let’s start with placing the resource vehicles: they deploy as a line around the resource point. On the left you can see that we have told this FARP that it should deploy 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:

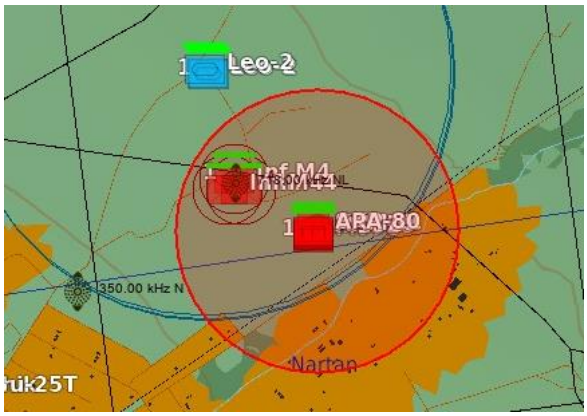




We follow the same procedure when placing the defenders that are spawned for the FARP when captured. The defenders always deploy inside a 100 meter radius, and you place the center of that circle with “rPhiHDef”. If the FARP is captured, the defending vehicles for the new owning faction use the same 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 they 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 that highly visible on maps, or only visible on the map when they are 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 (**note:** ‘hiding’ will *not* remove the FARP itself from the map, now will it make objects inside the game invisible; it will merely not draw the zone’s circle on the F10 map). These attributes all default to ‘false’, resulting in a FARP zone 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 detect 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.

5.3.19.2 Dependencies
Required: dcsCommon, cfxZones

5.3.19.3 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
refresh	The number of seconds between re-drawing all FARPs on the map. Necessary only for missions designed to run on multiplayer servers to work around a visibility bug for late-joining players. Set to -1 to disable. Defaults to -1 (disabled)

5.3.19.4 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

Name	Description
	around which the **resource** vehicles deploy as a line. Defaults to 0, 0, 0
redDefenders	typeStrings of defender vehicles. Example “ZSU-23-4 Shilka, ZSU-23-4 Shilka”. Defaults to “none” Special encoding: “none” – no vehicles
blueDefenders	typeStrings of defender vehicles. Example “Roland ADS, Roland Radar, Roland ADS”. Defaults to “none” Special encoding: “none” – no vehicles
formation	Formation of the defenders group. See dcsCommon for supported group formations. Defaults to ‘circle_out’.
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.
neutralProduction	Set to TRUE if you want a NEUTRAL FARP to produce units. Since neutral units do not fight, you normally don’t want this. Defaults to FALSE (no units produced if FARP belongs to NEUTRAL)

5.3.19.5 Persistence

FARP Zones persist the status of defenders and resource vehicles. Due to the way that FARPs interact with ground forces in the game, you may see a ‘FARP aligned with data’ message shortly after loading a mission, indicating that the FARP now should be available to your side (important when using SSBClient that prevents spawning from enemy FARP)

5.3.19.6 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.

5.3.20 mapMarkers

5.3.20.1 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).

Since the advent of “Daw Tools” in Mission Editor in 2022, this module has become redundant and will be removed.

5.3.20.2 Dependencies

Required: dcsCommon, cfxZones

5.3.20.3 Module Configuration

No special configuration required.

5.3.20.4 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”

5.3.20.5 Using the module

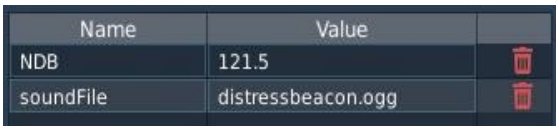
To enable, add the script to the mission as a DOSCRIPT action during Mission Start

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.

5.3.21 NDB

5.3.21.1 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 ADF 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 move with the unit. In the example to the right, we have linked an NDB at 540 kHz to the naval unit named 'Cruiser.'



By default, a moving) NDB updates its location once 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. To optimize performance, the NDB will only update if the difference in location between the linked unit's location and the NDB exceeds the module's config zone's "*maxDist*" attribute, which defaults to 50 meters (150 feet).

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

Turning a transmission on and off

When a mission starts, an NDB also immediately starts transmission. You can use the on? and off? inputs to use flags to turn an NDB's transmission on and off.

If you want to start the mission with the NDB not transmitting, use the "paused=true" attribute. You can then use the on? input later in the game to start the transmission on the NDB.

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 When the mission starts up, an NDB starts transmitting unless a "paused=true" attribute is present
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

5.3.21.2 Dependencies
NDB requires dcsCommon and cfxZones.

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

5.3.21.3 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
maxDist	In meters. Only relevant for NDB that are linked to moving units. When updating an NDB's location, it is only updated if the new location's distance to the current NDB's location is greater than this value. Defaults to 50 (meters)

5.3.21.4 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) The NDB starts transmitting at mission start unless there is also a paused=true attribute present (see below) **MANDATORY**
fm	If true, the transmission is in FM, else in AM Defaults to false (AM)
soundFile	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 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.
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"

5.3.21.5 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.

5.3.22 messenger

5.3.22.1 Description

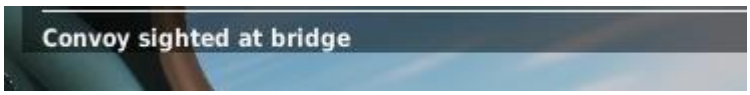
Name	Value
cloner	
in?	doClone
messenger?	doClone
message	Convoy sighted at Bridge

Originally created for convenience, and expanded for perfection, the messenger module is a massively enhanced replacement for ME's MESSAGE TO XXX action. Messengers have built-in interactive logic (so-called 'wildcards') to access world data, information about the zone that they are attached to (location, altitude, zone name), give live information about units (speed, heading) and can use the group/unit that they send their messages to as reference to calculate advanced information like bearing to a unit, closing velocity or aspect. Messenger supports the civilized world's standard (SI) unit system, plus the legacy "imperial" system (knots, feet, and other silly units).



Messengers also provide a simple mechanism to provide multiple possible responses, and then allow flags or other methods to select the appropriate response at runtime.

Messengers can display a message and/or play an audio file to the entire world, a coalition, a group, or single unit when triggered. Since DML modules are usually triggered when an input value changes, they are easier to set up for repeated uses, and dramatically easier to use.

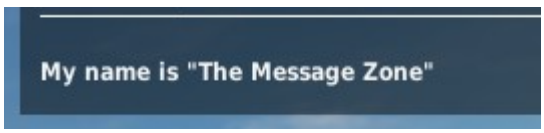


Better, and unlike ME's MESSAGE TO XXX action, a messenger module can be dynamically enabled and disabled by flags, making it very easy to create message controlling logic in your mission (for example simulating a unit that stops broadcasting text/audio messages when it is destroyed).

MESSAGE WILDCARDS

Name	Value
messenger?	whoAreYou
message	<n>My name is "<z>"<n>

Key to accessing live information with messenger is its powerful "wildcard" system: wildcards are text that is **replaced at run-time**, allowing you display current information that is not available at the time when you design the mission: Time of day, the current value of flags, the position of a zone (when this zone is moving), distance or bearing to units, etc. There are also 'convenience' wildcards to output a zone's name, current time, and line feeds



Convenience Wildcards
These wild cards display constants, information about the messenger zone itself, or the current time.

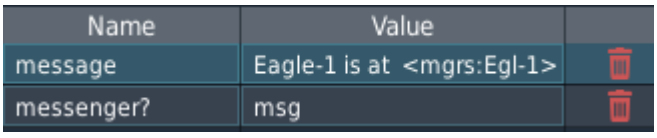
- <n> creates a line feed
- <z> – this is replaced with the zone's name as given in ME
- <t> – this is replaced with the current mission time in the format given in the zone's "timeFormat" attribute (which defaults to "HH:MM:SS"), e.g. "08:42:11"
- <lat> the latitude of the messenger zone's current position
- <lon> the longitude of the messenger zone's current position
- <ele> the elevation of the messenger zone's current position
- <mgrs> the zone's current position in MGRS coordinates

Wildcards that can access other Data:
These wildcards can access other flags, zones or units, and interpret the data they access there in some advanced form

- *Flag Values*
  To access a flag's current value from within a message, use the <v: flagName> wildcard, with flagName being the name of the flag that you want to access. For example, to access the flag 'killsA', you'd use the wildcard <v: killsA> in your message: "Player A has <v: killsA> kills"

- *Flag as Time Value*
  Likewise, you can access flags and have their value interpreted as time (in seconds), and use the zone's timeFormat attribute to convert that into a meaningful value. To access a flag and have the value converted to a time value, use <t: flagName>. Remember to set the zone's timeFormat attribute to the format that you prefer.

- *Accessing Positions: lat, lon, ele, mgrs* ***for zones and units***
  You can have messenger access a zone or unit's latitude, longitude, elevation, or mrgs by using the appropriate wildcards with a reference to the zone's or unit's name. If no unit or unit matches the reference, the entire reference is replaced with the (optional) messageError attribute (usually nothing).





  Note that messenger first looks for zones and only if it did not find a zone matching the name it will look for units. Unit and Zone names must match exactly.

- *Accessing map-named locales (town names) for units and zones: loc, twn*
  Messenger can substitute the name of the nearest map location (usually village or town as marked on the Map) with the <twn: xyz> wildcard, or substitute a location relative to the nearest



map location (i.e. “1.2km SW of Zugdidi”) with the <loc: ijkm> wildcard.
Requires that you have added the ‘twn’ module to the mission, else the wildcard will be replaced with “!twn!”.

- *Accessing coalition for flags, zones and units: coa*
  Messenger can access and interpret flag values, zone ownership, and unit coalition as faction names. By using <coa: abcd>, messenger first looks for a unit, then a zone, and finding neither, a flag named “abcd”. If it finds the unit named “abcd”, it retrieves the coalition that the unit belongs to, if if finds a zone named “abcd”, it accesses the owner of the zone. If it finds neither, it accesses the flag “abcd” and load that value. The value is then converted into either “NEUTRAL”, “RED”, “BLUE” or “UNKNOWN” (if the coalition value is neither 0, 1 or 2).

- *Accessing other Attributes: vel, alt, hdg, type for zones and units*
  Messenger can also access a unit’s or zone’s velocity, altitude or heading. Accessing these values for zones only makes sense if the zone is linked to a unit, and will then return the linked unit’s values.
  The following wildcards are defined

    - *vel* – returns the unit’s / zone’s velocity (in km/h or knots, determined by the value of messenger’s ‘imperial’ attribute). An unlinked zone always returns a velocity of 0

    - *alt* – returns the unit’s / zone’s altitude (in ft or m). When alt is above 1000, the altitude is rounded down to the nearest 100. An unlinked zone returns the height of the ground

    - *hdg* – returns the unit’s current heading in degrees. An unlinked zone returns zero.

    - *type* – returns the unit’s type (e.g. “A-10C”). When accessing a zone, the type is always “Zone”

    - *player* – returns the unit’s player name (e.g. “New Callsign”) as they are logged into the server or single-player mission. If the unit is not a player-controlled unit, the name is “Unknown”



Wildcards that return “here/there”-relative information
One particularly powerful ability of messenger is that it can derive unit-relative information like bearing from one unit (“here”) to another (“there”). This works only when messenger knows both parties (“here” and “there”). “There” is always the unit/zone that is referenced in the wildcard’s right side inside the message. For example, in <rng: Roosevelt>, the unit “Roosevelt” is “there”.

“Here”, on the other hand is derived from the messenger’s attributes:

- the *group* as defined with the attribute (in this case, “here” is the group’s lead unit). Message receivers are all members of the group.
- the *unit* as defined with attribute (“here” is the unit given). Message receiver is the unit only
- when neither group nor unit attribute is present, and the messenger’s zone is linked to a unit, that *linked unit becomes “here”*. Message receiver is the linked unit’s coalition (the normal use case is an observer unit that relays relative information about the “there” unit to everyone)

When messenger knows “here” and “there”, messenger allows additional wildcards:

- *bea – bearing*
  bearing from the “here” unit to the target zone or unit (“there”)

- *rng – range*
  range from “here” unit to the target unit

- *asp – aspect*
  Aspect of “there” unit towards “here” unit.

- *cls – closing velocity*
  How fast “here” and “there” are moving towards each other in km/h or knots. A negative value indicates that the units are moving apart

- *pcls – precision closing velocity*
  Like cls, but given in m/s or feet/second, with one decimal precision

- *clk – clock position*
  The ‘clock position’ of “there” unit relative to “here” unit’s current heading. “12” is straight ahead, “6” is behind

- *hnd – which hand*
  The “hand” position of “there” unit relative to “here” unit’s current heading. Possible responses are “ahead”, “right”, “behind” and “left”

- *sde – which side (‘nautical hand’)*
  Like hnd, except the possible responses re “ahead”, “starboard”, “aft”, “port”

- *rbea – response by bearing*
  “winds” the responses as given with the response attribute around the compass clock, and chooses the response based on the bearing to “there” unit as seen from “here” unit. see

Wildcards that interpret a flag’s value or unit’s existence

Messenger (and other modules that allow wildcards) can interpret the value of a flag (is it 0 or any other value) or existence of a unit (does it exist in the game or not) in various ways: There are convenience functions that automatically supply “yes” or “no”, “true” or “false”, “in” or “out”, and “alive” or “dead”. One general version of these wildcards allows you to supply the values yourself:

The convenience wildcard have the format <what: flagOrUnitName>, with “what” being one of the following:

- *yes – “no” or “yes”*
  say “no” when flag is zero or unit doesn’t exist, and “yes” otherwise. For example `"Goal reached: <yes: done>"` will translate to “Goal reached: no” if the value of the flag “done” is 0 (zero) and there is no unit named “done” existing in the game, and to “Goal reached: yes” otherwise (such a unit exists, or the flag’s value is anything other than zero.

- *alive – “dead” or “alive”*
  Similarly, the values that replace the wildcard are “dead” (flag is zero and no unit with that name exists) or “alive” otherwise

- *in – “out” or “in”*
  The same except we now receive “out” for zero/no such unit and “in” otherwise

- *“true” – “false” or true”*

The generalized version has the format

```
<A/B flagOrUnitName [zero return | any other return]>
```

With flagOrUnitName being the flag or unit to check. The two values that can be returned must be enclosed in square brackets “[]”, and are separated by the vertical bar “|”. So to create a wildcard that returns the value “bad” when flag “playerScore” is 0, and “good” when it anything else, we would write

```
<A/B: playerScore [bad | good]>
```

WILDCARD UNITS: SI OR IMPERIAL

Many wildcards return information about speed, height, velocity etc. You can use the zone’s attribute ‘imperialUnits’ to control if these are calculated/returned in a sensible form (when imperialUnits is omitted or set to false, messenger uses SI: km/h, km, m)

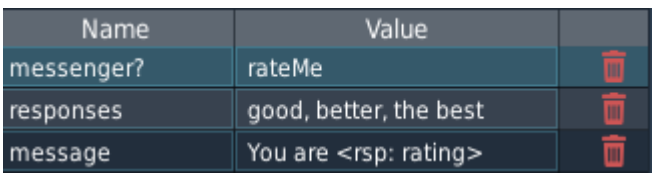


or otherwise (imperialUnits is set to true, in which case messenger uses knots, feet, nautical miles, tablespoon and whatnot):



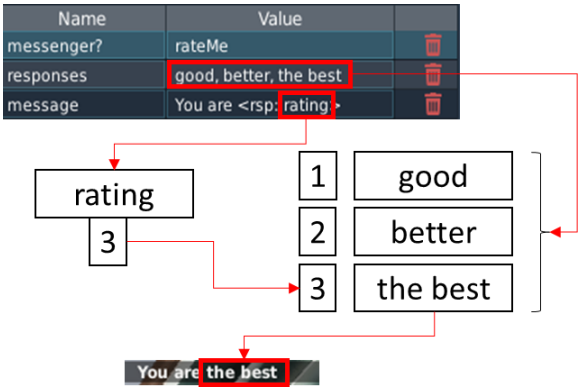
SELECTING RESPONSES

Each messenger allows you to pre-define multiple “canned” responses that can be selected from during runtime. The “responses” attribute sets this up: you list the possible responses, separated by comma.



Mission designers then have multiple methods to select these responses dynamically by choosing the appropriate wildcard:

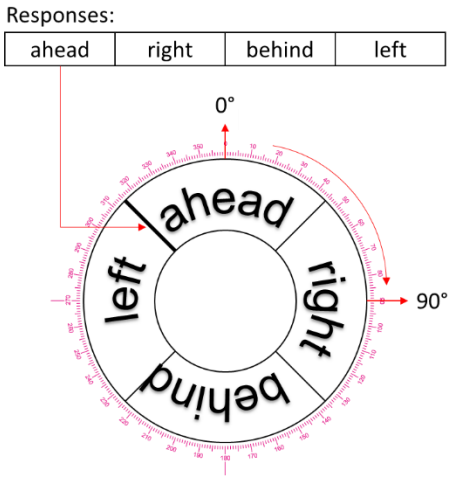
- *<rsp: flagName>*
  Uses the current value of the flag flagName as an index into the responses. The responses in the example to the right are numbered 1 through 3 and are “good”, “better”, and “the best”. So, if the flag “rating” currently holds the value 3, the third response is selected from “responses” and becomes the wildcard value. In our example that would be “the best”



- <rrnd>
  Randomly selects one of the responses each time that the messenger is triggered. Use it to make responses a little more random and introduce more polish to your canned responses.

“WINDING” RESPONSES AROUND THE COMPASS

One unique (and immensely useful) ability is that messenger can ‘wind’ the responses around the compass, dividing the compass rose into as many sectors as there are answers in responses. This is intended for selecting responses depending on the direction that is returned from the wildcards that can access this way of response selection.



Note that winding maps the responses around the compass rose in a way that respects the way we would usually subdivide a circle for directions. While it starts at 0° the sectors map ‘backwards’ to close the circle. If you, for example, map the two responses ‘forward’ and ‘backwards’ onto the compass rose, the first entry reaches from -90° to +90° to fill the top arc (i.e. it won’t start at 0° and run to 180°)

Currently, the following wildcards support winding responses around the compass:

- <rhdg unitName>
  Maps the responses by the heading that unit unitName is currently on.

- <rbea unitName>
  Maps the response by the bearing to unitName as seen from the unit that receives the message (if you target a group, all units in the group receive the same message, and the message is mapped according to the group’s lead unit).

TIME FORMAT
When a message displays a time value (via <t> or <t:name>), that time value is formatted according to the format that you can specify with the “timeFormat” attribute. By default, that format is "<:h>:<:m>:<:s>" which translates into 24-hour, leading-zero Hours:Minute:Second, e.g. “08:15:02”. When you pass a flag as time value, that flag’s value is always interpreted as seconds and formats the time accordingly

Messenger understands the following time formats

- <s>
  the time value in seconds. E.g., if the time value is 254, <s> is replaced with ‘254’ (no change to the value)

- <m>
  the time value in seconds as whole minutes. E.g., if the time value is 254, <m> is replaced with ‘4’, while 3891 returns “64”

- <h>
  the time value in seconds as whole hours. E.g., if the time value is 3891, <m> is replaced with ‘1’

- <:s>
  the time value converted to a “seconds in a minute” time value (0-60) and formatted with leading zero. E.g., if the time value is 64, <:s> is replaced with ‘04’

- <:m>
  the time value (in seconds) converted to “minutes in an hour” time value (0-60) and formatted with leading zero. E.g., if the time value is 64, <:m> is replaced with ‘01’, and 803 returns “13”

- <:h>
  the time value (in seconds) converted to a hours and formatted with leading zero. E.g., if the time value is 3764, <:h> is replaced with ‘01’

Accordingly, if you wanted to display a flag named ‘timeLeft’ as a count-down value showing remaining minutes and seconds (i.e. “73:55”), you’d first configure timeFormat attribute in the zone to be “<m>:<:s>” and then use the wildcard <t: timeLeft> in your message

MESSAGE RECEIVERS (everyone, coalition, groups, units)
You can control who receives the message sent by messenger on four levels:

- If you don’t specify anything, the message is broadcast to everyone
- If you add a ‘coalition’ attribute, the message is only broadcast to players on that side
- If you add a ‘group’ attribute, only units in the groups listed (e.g. “`Eagles-1, Hogs-3`” receive the message. **Enables unit-relative wildcards**. All unit-relative wildcards resolve to the position of the group’s lead unit (unit 1)
- If you add a ‘unit’ attribute, only the unit listed here receive the message. **Enables unit-relative wildcards**.

Note that the attributes '*coalition'*, '*group*' and '*unit*' are all mutually exclusive. You can at most have one of the 'coalition', 'group' or 'unit' attribute per messenger. If you add more than one receiver attribute, the results aren't defined and will prompt a warning from the messenger module during mission start-up.

TURNING A MESSENGER ON AND OFF

Just like a radio or telephone, you can turn a messenger on and off. When 'off' it won't react to signals on messenger?. Use the messageOn? And messageOff? inputs to turn the messenger on and off.

When a messenger starts, it's on by default, and you can start it up by setting the onStart attribute of a messenger to 'false'.

5.3.22.2 Dependencies

Messenger requires dcsCommon and cfxZones to run

5.3.22.3 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

5.3.22.4 ME Attributes

Name	Description
messenger?	Watchflag. When triggered, the module will display the message and/or play sound. **MANDATORY**
message	The text of the message to be displayed **FORMATTING WILDCARDS** • <n> creates a new line • <z> is replaced with zone's name • <t> is current time in the format as defined with the *timeFormat* attribute **RESPONSE-SELECTION WILDCARDS** • <rsp: flag name> looks up the value of <flag name> and uses that value and an offset into the responses given with the 'responses' attribute. The first response has an index of 1. If the flag's value is less than 1, the first response is returned, if the value is higher than the number of responses, the last response is returned.

	• <rrnd> randomly selects one of the possible responses and returns that response • <rhdg: unit/zone> wraps responses around the compass, and then uses the unit/zone's heading as offset. Zones only have a heading if they are linked to a unit. [not yet implemented] • <rbea: unit/zone> selects and returns one of possible responses using the bearing (in degrees) from unit/group to unit/zone as offset into *responses.* [requires group or unit attribute be set] **DATA ACCESS WILDCARDS** • <lat> the latitude of the zone's current position • <lon> the longitude of the zone's current position • <mgrs> the zone's current position in MGRS coordinates • <v: flagName> is replaced with the value currently held by the flag *flagName* • <t: flag name> uses the value from flag *flag name* and interprets it as a time value, formatted according to the timeFormat attribute • <lat: unit/zone name> outputs the latitude of the zone or unit that matches the name *unit/zone name*, or "messageError" (usually an empty string) if neither can be found • <lon: unit/zone> outputs the longitude of the zone or unit that matches the name *unit/zone name*, or "messageError" (usually an empty string) if neither can be found • <ele: flagName> outputs the elevation of the zone or unit that matches the name *unit/zone name*, or "messageError" (usually an empty string) if neither can be found. Elevation is calculated in meters (default) or feet if the "imperial" attrigute is true • <latlon: unit/zone> outputs the latitude and longitude of the zone or unit that matches the name *unit/zone name*, or "messageError" (usually an empty string) if neither can be found • <lle: unit/zone> outputs the longitude, longitude and elevation of the zone or unit that matches the name *unit/zone name*, or "messageError" (usually an empty string) if neither can be found • <twn: unit/zone> outputs the name of the nearest named location on the map. Requires the twn module • <loc: unit/zone> outputs position relative to nearest named map location. Requires twn module • <mgrs.: flagName> outputs the mgrs. coordinates of the zone or unit that matches the name *unit/zone name*, or "messageError" (usually an empty string) if neither can be found • <vel : unit/zone> outputs the velocity (in km/h or knots, depending on imperialUnits) of unit. Zones only have a velocity if they are linked to a master unit • <hdg: unit/zone> outputs the direction that the unit/zone referenced is heading. Zones only have a heading if it is linked to a master unit, and the heading then returned is the one of the linked unit.

	• <alt: unit/zone> outputs the altitude (barometric) of the unit/zone. If the zone is unlinked, it returns the altitude of the land at the zone's center. If the zone is linked to a unit, it returns the altitude of that unit. Altitude is returned in meters or feet, depending on the imperialUnits attribute. • <type: unit/zone> returns the type of the unit (e.g., "A-10C"). For zones, the type returned is always "Zone" • <player: unit> outputs the player's log-in name if that unit is controlled by a player (e.g., "New Callsign"), if the unit isn't controlled by a planer it returns "Unknown" • <coa: flag/unit/zone> outputs the faction (e.g., "RED") of the flag/unit/zone. It returns the first name match it finds: Unit before zone before flag. **"HERE/THERE" RELATIVE (requires group or unit)** These wildcards are only available when you have supplied messenger with information about "here" – with a *group*, *unit* or *linkedUnit* attribute. "There" is the unit that is referenced in the wildcard. • <bea: units/zone> bearing (in degrees) from "here" to "there" • <rng: unit/zone> range (distance) from "here" to "there" • <clk: unit/zone> direction as "o'clock" to the unit/zone as seen from the player unit/zone's heading (12 is straight ahead, 6 is behind) • <hnd: unit/zone> direction "which hand" to the unit/zone: "ahead", "right", behind", "left" as seen from "here" • <sde: unit/zone> direction "side" to the unit/zone as seen from "here": "ahead", "starboard", "aft", "port" • <asp: unit/zone> aspect of "there" towards "here". Returns "hot" / "beam" / "drag" • <cls: unit/zone> closing velocity of "here" and "there". A negative closing velocity means that distance is growing. Closing velocity is given in km/h or knots, depending on imperialUnits • <pcls: unit/zone> precision closing velocity of "here" with "there". Closing velocity is given in m/s or ft/s with up to one decimal, e.g., "1.3" **Interpret values** Return one of two possible values, based on the flag's value or existence of the unit named • <yes: flagOrUnitName> returns "no" if flag's value is zero and no unit of that name exists, "yes" otherwise • <true: flagOrUnitName> returns "false" if the flag's value is zero and no such unit exists, "true" otherwise • <in: flagOrUnitName> returns "out" if the flag's value is zero and no such unit exists, "in" otherwise • <alive: flagOrUnitName> returns "dead" if the flag's value is zero and no such unit exists, "alive" otherwise • <A/B: flagOrUnitName [zero val | other val]> returns whatever is inside the square brackets "[]" and left of the vertical bar "|" ("zero val" in this case) when the flag's value is zero and no such unit exists, and whatever is to the right of the vertical bar

	"|" inside the square brackets "[]" otherwise
responses	A list of comma-separated possible responses that can be accessed by various wildcards. Note that the responses themselves must not contain a comma ",". Example: "good, better, the best" is treated as three separate possible responses: "good", "better" and "the best". Defaults to <none>
triggerMethod msgTriggerMethod	Defines the trigger condition for DML Watchflags. Use only one synonym per zone Defaults to "change"
clearScreen	If true, erase all existing messages. Defaults to false
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 should receive the message/sound. If no coalition is given, text and sound are played to all. Legal values are "red", "blue", "neutral", 0, 1, 2 Note that if given, the attributes 'coalition', 'group' and 'unit' are mutually exclusive. Defaults to <none>
group msgGroup	The name of the Groups (separated by comma ',') that should receive the message/sound. Adding a group attribute enables unit-relative wildcards. Note that if given, the attributes 'coalition', 'group' and 'unit' are mutually exclusive. Defaults to <none>
unit msgUnit	The name of the Units (separated by comma ',') that should receive the message/sound. Adding a group attribute enables unit-relative wildcards. Note that if given, the attributes 'coalition', 'group' and 'unit' are mutually exclusive. Defaults to <none>
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 was already turned off, nothing happens. Defaults to <none>
mute messageMute	If set to true, the messenger starts muted and requires a signal on messageOn? To activate. Defaults to false
duration messageDuration	Time (in seconds) how long a message should stay on-screen Defaults to 30 seconds

timeFormat	Time format for any time values Defaults to “<:h>:<:m>:<:s>” – standard 24 hour time
imperial imperialUnits	When true, elevation is calculated in feet (imperial units) else meters. Defaults to false (meters)
error messageError	The text to substitute for a unit or zone reference if the units or zone cannot be found. Defaults to “” (empty string)

5.3.22.5 Using the module

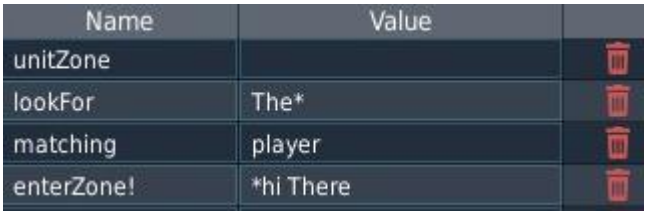
To enable, add the script to the mission as a DOSCRIPT action during Mission Start

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

5.3.23 unit Zone

5.3.23.1 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 for 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 group 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 classic 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 unitZone 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.

  NOTE
  unitZones "lookFor" defaults to "*" – the wildcard asterisk without any leading characters. This means that it matches any name.

  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.

5.3.23.2 Dependencies
unitZone requires dcsCommon and cfxZones to run

5.3.23.3 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

5.3.23.4 ME Attributes

Name	Description
unitZone	Marks this ME Zone as an anchor for unitZone. The value of this

Name	Description
	attribute defines which coalition groups/players are checked. Legal values are “0”, “1”, “2”, “red”, “blue”, “neutral”. Note that “**0” (zero) and ‘neutral’ means ‘both’:** unitZones never considers groups belonging to the neutral coalition. Defaults to 0 (both coalitions red and blue) **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 accepted, e.g. if you supply ”Hel*” all of the following would be accepted: • Hel • Hello World • Helo Rescue-1 • Hellfire If you want to match all groups or players, simply supply “*” (default) as this will match all names. 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. Defauls to “*” – meaning all names are matched.
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
filterFor filter	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 Defaults to <none>
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) Defaults to <none>

Name	Description
changeZone!	Changes this flag whenever enterZone! or exitZone! are triggered Defaults to <none>
method uzMethod	DML Flag method for output. Use only one synonym per zone Defaults to “inc”
uzOff?	Watchflag. When triggered, this zone will no longer perform checks. When already off, nothing happens Defaults to <none>
uzOn?	Watchflag. When triggered, this zone will resume checks. When already on, nothing happens Defaults to <none>
triggerMethod uzTriggerMethod	Method that determines when the watchflags should trigger. Default is “change”
uzDirect uzDirect# direct#	Used mainly to control (directly enable/disable modules and open/close gates (changer modules)) 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>
uzDirectInv uzDirectInv# directInv#	Like uzDirect, just inverted (meaning that when uzDirect is 0 this value is 1 and vice versa): This is provided bevause unitZones is often used to control many functions (especially gates), and this saves an inverter (changer) module: • 0 when one or more units in the zone • 1 when none of the indicated units in the zone. Default is <none>

5.3.23.5 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.

5.3.24 groupTracker

5.3.24.1 Description

GroupTracker is a module that allows you to generate DML-style events whenever a group or set of groups is destroyed. If you have some ME experience, look 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 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 WORKS

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 those groups that you add if they don't match the categories that you choose.
- 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 output "numGroups" and the total number of units tracked to the "numUnits" output

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
  Triggered with method 'inc' whenever a group is added. Therefore contains the total of all groups added to this tracker (this can be much higher than the number of groups currently tracked)

Name	Value	
tracker		
removeGroup!	*bdead	
messenger		
message	Red Kills: *value	
messageOut?	*bdead	
messageValue?	*bdead	

- removeGroup
  Triggered with method ‘inc’ whenever a group is removed (dead). Therefore contains the running total of the number of groups that were tracked by this tracker and have been destroyed up to now

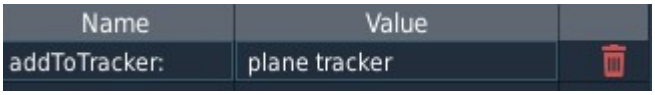
In effect, groupTracker not only allows you to be notified when a group is added to the tracker or a tracked group is destroyed, it also always gives you access to totals, and these are most often used to control scenario/victory condition or mission states. By providing the ability to count from a set,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

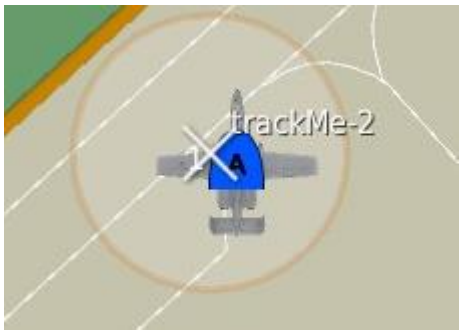
- Provide a kill count
- Provide a ‘units left’ count
- Trigger reinforcements if 3 groups from a set of groups have been destroyed
- Disable spawners/cloners when a maximum number of groups is reached and re-enable them when it falls below a minimum
- Trigger if all tracked groups are destroyed
- Destroy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

Due to the way it functions and integration with cloners, you can use a groupTracker as an easy add-on for cloners to track groups that were created by multiple spawners of spawn cycled as it’s allGone! output is the cloner’s empty! analogue, and a tracker’s destroy? Input works like a cloner’s despawn? input.

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.

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.

Similarly, the destroy? Input can be used to destroy all tracked groups. This mimics a cloner's despawn? input and can be used in conjunction with a spawner to destroy all clones that were spawned over multiple clone cycles.

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	Description
addGroup!	Whenever a group is added to the tracker, the value of this flag is changed according to the chosen DML method.
removeGroup!	Whenever a tracked group is destroyed, the value of this flag is changed according to the chosen DML method.
numGroups#	The value of this flag always represents the number of groups currently watched by this tracker. Its value is updated 1/ups times per second.
numUnits#	The value of this flag always represents the total number of units currently watched by this tracker. Its value is updated 1/ups times per second.

Name	Description
allGone!	When the number of tracked groups falls to zero, this flag is banged. If it was zero before, no output signal is generated.

5.3.24.2 Dependencies
groupTracker requires dcsCommon and cfxZones

5.3.24.3 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

5.3.24.4 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.

Name	Description
	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 changed according to the chosen DML method. Defaults to <none>
removeGroup!	Whenever a group is removed from the tracker, the value of this flag is changed according to the chosen DML method. 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>
numUnits#	The value of this flag always represents the total number of units 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
triggerMethod trackerTriggerMethod	Watchflag method for inputs Defaults to 'change'
destroy?	Watchflag that when triggered destroys all groups that are currently being watched. If any groups are destroyed, the removeGroup output is increased by the number of groups that were removed. numGroup is set to 0 Defaults to <none>
method trackerMethod	Method to bang! on output flags Defaults to "inc"
allGone!	Flag to bang! when the number of tracked groups falls to zero. If it was zero before, no output signal is generated.

5.3.24.5 Persistence

Group Tracker supports persistence. Note that due to the way that persistence works, the numGroups and numUnit outputs are set one second after all groupTracker loads (in a process called 'late bind') to allow later-loading modules to spawn the units that groupTracker tracks. If you trigger on those outputs, make sure to persist the output flags with persistence's "saveFlag?"

5.3.24.6 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.

5.3.25 wiper (Unit/Object/Debris removal)

5.3.25.1 Description

Wiper (with a 'w') is a module that can, upon activation remove specific objects that are inside a 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 various objects that populate the map out-of-the-box. And no, trees are (sadly) *not* part of that definition. As aren't some 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

IMPORTANT
For reasons unknown, DCS keeps a separate category of objects that are created when things explode: craters, flames, smoke, debris and wrecks. Wiper can also remove some of these, and it uses a separate ability "declutter" to do this. Please see "DEBRIS AND WRECKS", below

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.

DEBRIS AND WRECKS

After units blow up or crash, they may leave behind some form of representation like craters, debris, wrecks, flames, or smoke. These have long been inaccessible to modules, but a recent change has allowed limited access to such artifacts. They can't be differentiated and removing them is a 'some or nothing' affair: Wiper can attempt to remove most of them inside a zone. To tell wiper to attempt removing any of these destruction artifacts inside a zone, set the 'declutter' attribute to true. Wiper will then try to remove as best it can whatever artifacts DCS allows to remove. Currently, this is limited to:

- Craters (although AI will still refuse to land if they were placed on a runway, AI resumes landings after one hour has passed.)
- Debris
- Wrecks

Most notably, currently wiper's 'declutter' ability *does not extend to fire nor smoke*. Since the flames are purely visual effects, and do not affect units walking through them, this is a visual nuisance only. Flames and smoke time out after a couple of minutes.

LIMITATIONS

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

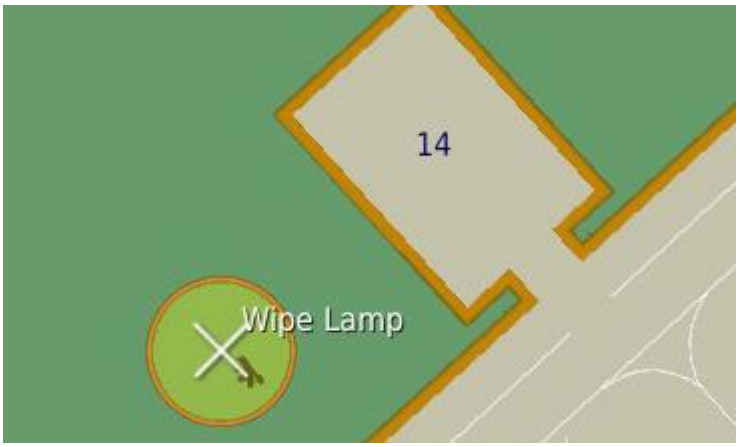
- When units fight, some may get destroyed. Depending on how overwhelming their defeat and how strong they are, the transition from living AI unit to dead husk may be different. Infantry getting killed usually transitions from alive to dead by playing a short animation cycle, and then replacing the live 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 remove some of the debris left behind directly, while some of the wreckage requires that you activate the 'declutter' attribute as well. Also, in current DCS versions, flames and smoke that erupt from vehicles are not part of the 'detectable' world, and therefore cannot not be removed by wiper modules. They will have to run their course (they'll disappear eventually)



- 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.

WARNING:
A wiper zone **must be circular** to work correctly. If your wipe zone is polygonal, the results can be (and will in the worst possible moment, of course) be unpredictable.

5.3.25.2 Dependencies
Wiper requires dcsCommon and cfxZones to run

5.3.25.3 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

5.3.25.4 ME Attributes

Name	Description
wipe?	Watchflag. Triggers a wipe cycle **MANDATORY**
triggerMethod triggerWiperMethod	Method that triggers the Watchflag Defaults to ‘change’
category wipeCat wipeCategory	A list, separated by comma “,” of object categories 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 If a category is not recognized, it defaults to “3” (static) A special value ‘none’ can be used to skip object removal entirely. This can be useful if you are only interested in the ‘declutter’ (wreck and debris removal) ability.

Name	Description
	Examples • "`none`" (skips entire object removal step) • "`unit, 3`" (wipes units and static objects) • "`cargo`" (wipes cargo type objects) Defaults to 'none'
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
declutter	If set to true will remove all debris, wrecks, craters and similar detonation artifacts from inside the zone. *Caution:* This option invokes DCS's `world.removeJunk()` method which is currently suspected of doing some nefarious stuff, including mission crashes in multiplayer. Suspected, not proven. Defaults to false (no debris removal)
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.

5.3.25.5 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.

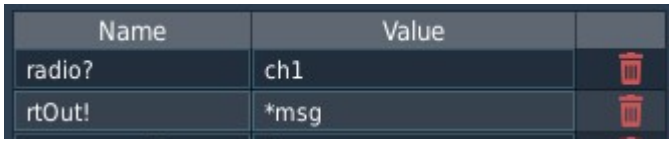
Then, to remove objects inside the zone, trigger the Watchflag. Note that you can trigger the wiper multiply times

5.3.26 radioTrigger

5.3.26.1 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.

5.3.26.2 Dependencies

Radio Trigger requires dcsCommon and cfxZones

5.3.26.3 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

5.3.26.4 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>

5.3.26.5 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

5.3.27 baseCaptured

5.3.27.1 Description

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

Note:
The *airfield* module provides very similar, and expanded abilities. Make sure to also check out that module and see which one fits your mission's requirements better.

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 its 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 to the intended base, it is your onus to ensure that there are no bases closer to it, 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).

CONTESTED FARP/AIRFIELD

By default, baseCaptured also supports the 'contested' state of a FARP or airfield when blue and red vehicles are close by. When a FARP becomes contested, the module sends out a signal on the contested! output.

OUTPUT

baseCaptured provides very different outputs:

- Creates a DML Flag change when the nearest base is captured. These are the baseCaptured! contested!, 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, 3 = contested). This is frequently used to count the number one side holds and generate actions when that number falls below, or rises above, a threshold.

5.3.27.2 Dependencies

baseCaptured requires dcsCommon and cfxZones

5.3.27.3 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

5.3.27.4 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’
blueCaptured! 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>
contested!	Flags to bang! when closest base becomes contested and belongs to neither blue nor red Requires handleContested be true in configuration (default) Defaults to <none>
baseOwner	Flag that is set by the module to the current faction (0 = neutral, 1 = red, 2 = blue, 3 = contested) that currently holds this base.

5.3.27.5 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 associated with the nearest base.
- Add other attributes as needed

5.3.28 radioMenu

5.3.28.1 Description

DCS allows you to add menus and sub-menus to the Communication→F10 Other… menu. The RadioMenu module is DML's equivalent, allowing you to easily install menus with up to four commands per menu that all drive (DML-Style) flags.

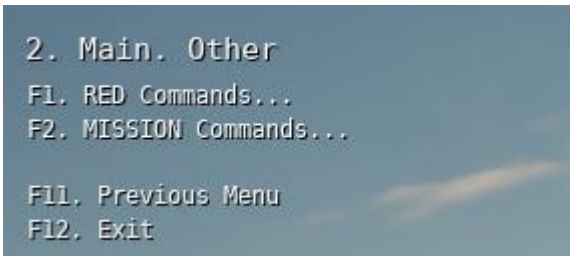
DML's radio menus are easier to set up than using plain ME, support a broader range of options (like type-specific menus, e.g. all "A-10A" player planes) and support cool-down and multi-use.

To give even more flexibility (should you run out of "menu space" in your mission), you can use the radioMainMenu ability to further group multiple radioMenus under one main entry.

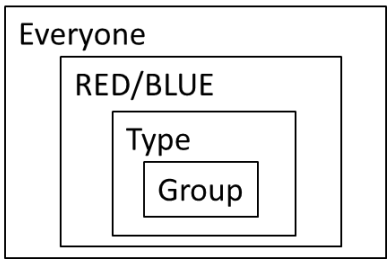
MENUS FOR ALL / COALITION / GROUPS / TYPES

By default, for each trigger zone that has a radio menu attribute, DML installs a menu in the Communication→F10 Other… menu [Note: this is radioMenus default "simple" behavior. DML supports grouping multiple radioMenus under another menu – for this plead see 'Grouping Menus', below]. In such a menu, you can have up to four separate *menu items*. DML supports type-, group- and coalition-specific menus, so you can restrict the availability of a menu.

The image on the right shows two menus installed by two different trigger zones with a radio menu attribute: the 'RED Commands' menu (which is only visible to the red side), and "MISSION Commands" that is visible to all players. Choosing either menu branches to the menu items available for that menu.



To control who can access a menu, you use the 'coalition', 'groups' and 'types' attributes as follows:



- If you omit all attributes, the menu is available to everyone
- If you add the coalition attribute alone, only that coalition can see the menu, if you choose 'neutral' or 0 as coalition, all sides get access to the menu
- If you add a 'groups' attribute, any information given in the 'coalition' attribute is ignored, and the groups listed get access to the menu
- If you add a types attribute, any player who crews a unit that matches one of the types (e.g. 'A-10A, UH-1H') listed receives access to the menu. You can narrow this by also providing the coalition attribute, so only players who match the coalition and type receive access to the menu.
- If you add both 'groups' and 'types' attributes, 'types' is ignored.

You can use the 'types' attributes to restrict access to a menu to generic and specific types as follows:

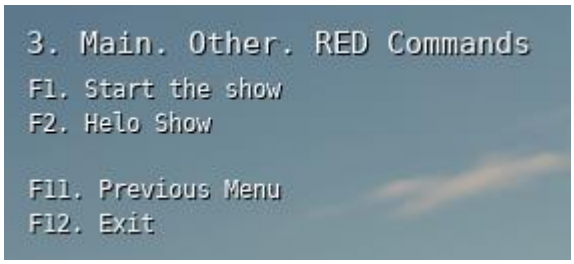
- ‘helo’ grants access to this menu to all players controlling helicopters
- ‘plane’ grants access to this menu to all players in fixed-wing aircraft
- Type names (e.g. “A-10A”) restricts the menu to all players who control a plane of exactly that type

The ‘types’ attribute supports comma-separated lists, so you can add multiple types. For example a ‘types’ attribute with “helo, A-10A” as value grants access to all helicopters and the A-10A.

MENU ITEMS, COOLDOWN, FLAG, VALUE
For each menu installed, you can add up to four menu items per menu.

When the player chooses an item, DML bangs the flags that are associated with that item, and initiates a cool-down timer. When the player chooses this menu item again before the cool-down has timed out, no flag is banged, and a message is displayed instead.

In the image to the right, we had previously chosen the “RED Commands” menu and are now presented with two items to choose from: “Start the show”, and “Helo show”. Choosing either item results in DML banging the flags associated with it and initiating the cool-down.



When an item is on cool-down, choosing it produces a message that optionally can also display the remaining time (in seconds)

ACKNOWLEDGE MESSAGE
When a player chooses a menu item, an optional “acknowledge” message can be sent out immediately. The message is defined per menu item individually via the ackX attribute, and it is visible to the same scope as the menu: group, faction, or all.

You can use wildcards to define dynamic acknowledge messages. In addition to some specialized wildcards, Radio Menu supports the same wildcards as messenger (**see the messenger module** for a more detailed description), for example

- <group> - name of the group that the player’s unit is in (only works for group-specific menus)
- <n> creates a line feed
- <z> - radioMenu zone’s name as set with ME
- <t> - current mission time in the format “HH:MM:SS”, e.g. “08:42:11”. Format be further configured with the *timeFormat* attribute.
- <lat> - latitude of the radioMenu zone’s current position
- <lon> - longitude of the radioMenu zone’s current position
- <ele> - elevation of the radioMenu zone’s current position.
- <mgrs> - radioMenu zone’s current position in MGRS coordinates

- <v: flagName> - value of flag <flagName>
- <t: flagName> - value of flag <flagName> interpreted as time.
- <lat: unit/zone> - latitude of unit/zone with that name

- `<lon: unit/zone>` - longitude of unit/zone with that name
- `<ele: unit/zone>` - elevation of unit/zone with that name. Format be further configured with the *imperialUnits* attribute.
- `<mgrs: unit/zone>` - mgrs of unit/zone with that name
- `<vel: unit/zone>` - velocity of unit/zone with that name. Format be further configured with the *imperialUnits* attribute.
- `<alt: unit/zone>` - altitude of unit/zone with that name. Format be further configured with the *imperialUnits* attribute.
- `<hdg: unit/zone>` - heading of unit/zone with that name
- `<type: unit>` - type (e.g. “A-10A”) of unit with that name
- `<player: unit>` - player callsign who controls a unit with that name

COOLDOWN MESSAGE

When a player chooses a menu item while a cooldown is active, no flag is banged, and a message is displayed instead. You can define your own message that is to be displayed, and access/display the current number of seconds until the item becomes available again. Radio Menu provides special wildcard characters that allow you to format and display the remaining time in an easy way.

You can use more than one wildcard. For example, “please wait <:m>:<:s>” will translate into “please wait 02:12” when the cooldown is 132 seconds left. Use the following wildcards (note that they are identical to messenger’s ‘display flag as time wildcards)

- <s>
  remaining cooldown in seconds. E.g., if remaining cooldown is 254, <s> is replaced with ‘254’

- <m>
  remaining cooldown as whole minutes. E.g., if remaining cooldown is 254, <m> is replaced with ‘4’, while 3891 returns “64”

- <h>
  remaining cooldown as whole hours. E.g., if remaining cooldown is 3891, <m> is replaced with ‘1’

- <:s>
  remaining cooldown converted to a seconds time value (0-60) and formatted with leading zero. E.g., if remaining cooldown is 64, <:s> is replaced with ‘04’

- <:m>
  remaining cooldown converted to a minutes time value (0-60) and formatted with leading zero. E.g., remaining cooldown is 64, <:m> is replaced with ‘01’, and 803 returns “13”

- <:h>
  remaining cooldown converted to an hours time value and formatted with leading zero. E.g., if remaining cooldown is 3764, <:h> is replaced with ‘01’

Additionally, the cooldown message supports the same wildcards as the acknowledge message

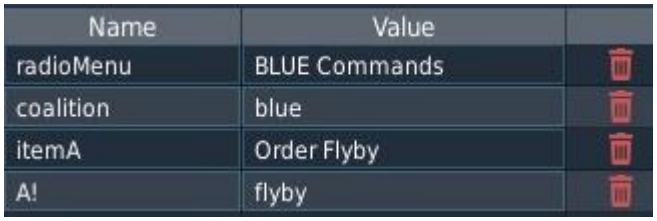
ADVANTAGES
Using DML Radio Menu in your mission opens several advantages of classic ME trigger-based menus:

- Easy to use, easy to modify
- More than one flag per menu item (default DML feature) can be changed
- Natural DML integration – multiple uses result in multiple flag changes (default DML 'Method' behavior). You can change this to single-use by setting a trigger zone's method accordingly
- Built-in cool-down period, and cool-down messaging (including remaining time)

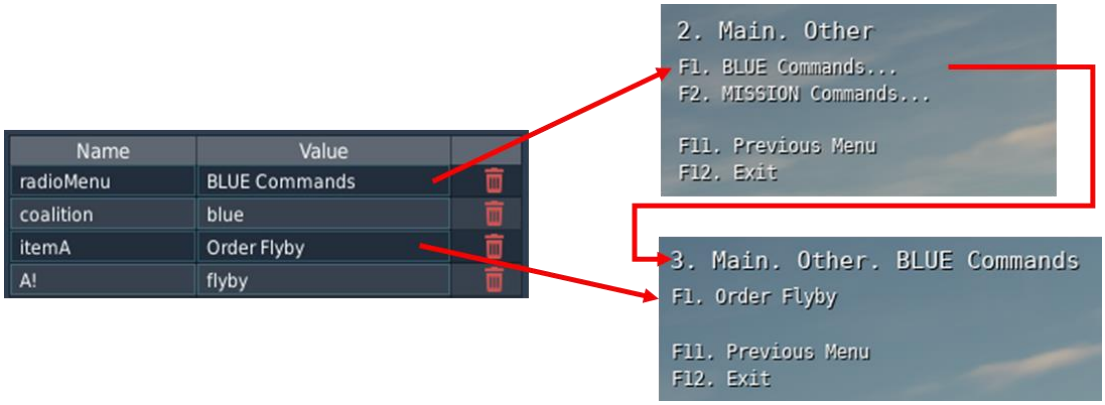
LIMITATIONS
To make them easy to use, RadioMenu imposes some limitations on their use:

- Items always install inside named menus.
  This means that to install a command on the 'Communications→F10 Other' top level, you will need to use another method.
- Limited to four (4) items per menu
- No group-specific menus, only global and coalition-specific

MINIMAL SETUP
To add a Communications→F10 Other menu to your mission requires minimal setup: simply add a trigger zone anywhere on the map (the location is not important), and add the following attributes:



- radioMenu: the value of this attribute appears as a new menu in the F10: Other menu.
- **OPTIONAL:** coalition controls which coalition (red/blue) will have access to this menu. If you omit this attribute, all sides can see it
- itemA (or itemB, or itemC or itemD) the command that appears when the player opens this menu. Currently, upt to four items (A-D) can be defined
- A! (or B! or C! or D!) the flags to bang according to the current DML method when the player chooses the corresponding item (by default, the fag's value is increased).



INDIVIDUAL OUTPUT METHODS/VALUES FOR A!, B!, C! AND D! - valX

By default, radioMenu uses the zone's output method to change/set the corresponding flag's value when a menu item is chosen. The method itself defaults to 'inc' (increment value), so unless you say otherwise, each time that a player selects "Order Flyby" in the above example, the value for the flag named 'flyby' is increased by one. This triggers connected modules multiple times, and is intentional.

This is in sharp contrast to how DCS itself handles radio menu items: in DCS you can only set a flag to a specific value when a menu item is chosen, for example the value "3". Each subsequent selection of that radio item will again set the connected flag to 3, and if it was set to 3 before, it is set to 3 again. DML modules will not see this as a change and will not trigger. But in some missions (especially when you support some legacy mission), it may be desirable to emulate that behavior.

To provide compatibility with DCS's rather pedestrian and similar use cases, radioMenu supports optional **menu-item individual** methods `outA`, `outB`, `outC` and `outD`, which you can use to control the *output method*. If you omit the *`outX`* for a menu item, that output flag, when chosen, is handled by the output method (usually 'inc'). If present, you can specify any legal DML method to set the outcome item-individually.

Examples:

- `#3` or `3`
  This sets the value of the flag to the number 3. This mimics DCS's standard radio menu behavior
- `pulse, 4`
  Uses the 'pulse' method with a 4 second "high" phase on the flag before setting it "low" again
- `Off`
  Sets the flag zo zero

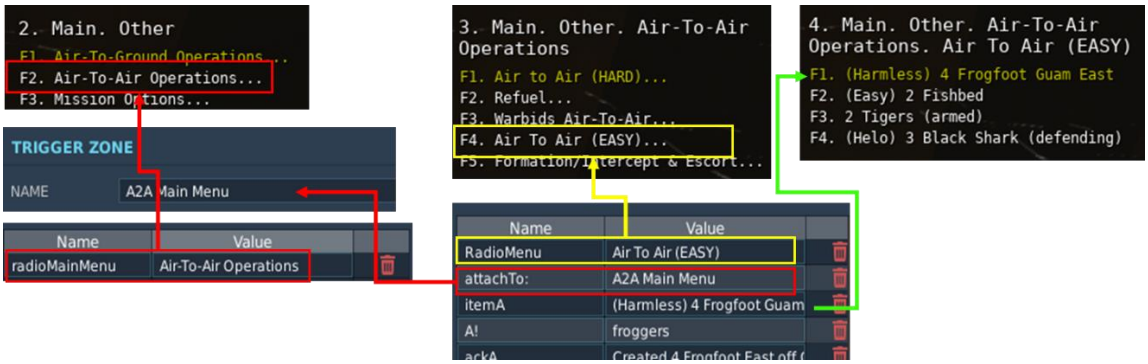
GROUPING MENUS

The simplicity and versatility of DML's radioMenu module has led to an interesting effect: a proliferation of menus in missions. Mission designers love the ability to add player options, and DML happily obliges. That is, until DCS doesn't. An undocumented "Feature" of DCS's API is that if you add more than 10 menus to "F10:Other", some of the menus disappear. "Some?" – yeah, some. It's unpredictable which will remain, and which will vanish.

Unless resolved with other means, this limits the number of user commands a mission designer can add to a mission with DML to 40: 10 menus with a maximum of 4 items each.

Enter *radioMainMenu*. Like *radioMenu*, this adds a new entry into F10: Other. The difference is that radioMainMenu does not have any items itself, it merely saves as a slot where a radioMenu can plug into. You do that by adding a single attribute to a *radioMenu*: "*attachTo:*" note the colon ":"). Doing so attaches the radio Menu to the *radioMainMenu* instead directly to F10: Other.

In other words, you can use *radioMainMenu* to easily group *radioMenu*s. So, by using simple grouping, the number of commands that you can allow players to issue grows from 40 to 10 x 40 = 400 (this assumes 10 *radioMainMenu* in Other, each with 10 *radioMenus* that have 40 items each).



Now, since *radioMainMenu* itself allows using the "*attachTo:*" attribute, there is no practical upper limit to the number of *radioMenu*s that you can add to your mission. As a rule of thumb, I recommend that you try and keep the number of "nestings" as low as possible.

MANAGING radioMainMenu's SCOPE (controlling menu access)
radioMainMenu supports the same filtering as radioMenu: you can restrict access to the menu by coalition, type, or group. This means that access restrictions that apply to a menu are added sequentially, and the scope of access only narrows, it never broadens.

Example:

- A radioMainMenu "Red Action" is restricted to RED side only via the *coalition* attribute
- A radioMenu "Plane Action" is restricted to planes only.
- If that radioMenu "Plane Action" uses attachTo: to plug into "Red Action", only players that are inside planes and are member of the red faction have access to that menu (radioMenu strips all non-planes, and then radioMainMenu strips everyone who is not on RED side)

5.3.28.2 Dependencies
Radio Menu requires dcsCommon, cfxZones and cfxMX

5.3.28.3 Module Configuration
To configure the RadioMenu module via a configuration zone,

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

Name	**Description**
verbose	Show debugging information. Default is false

5.3.28.4 ME Attributes

Name	**Description**
radioMenu	Name of the menu to install in the Communications→F10 Other menu or the *radioMainMenu* that you refer to in *attachTo:* (see below). **NOTE:** The same trigger zone **must not** have both a radioMenu and radioMainMenu (see below) attribute. **MANDATORY**
attachTo:	By default, a radioMenu is added to a mission’s ‘F10: Other’ menu. You can use radioMainMenu zones (see below) to group radioMenus. If you use “attachTo:” (mind the colon “:”), this radio menu attaches to the radioMainMenu that is described as value for this attribute. The trigger zone that you refer to as value must exist, and have a radioMainMenu attribute. Defaults to <none> (radioMenu is installed into F10-Other)
coalition	The coalition that has access to this menu. If omitted or set to ‘neutral’, *all* coalitions have access. ‘blue’ or ‘red’ restricts access to that coalition. Defaults to <no coalition>
group groups	Restricts this menu only to groups with that name. Supports comma separated groups names, e.g. “Eagles 5, Uzi One, Aleph” will make this menu available to all members of those groups. **Supports wildcard** names: If you supply a group name “Cat*”, all groups **that start with** “Cat” (WARNING: CASE SENSITIVE!) get access to this menu. For example, groups name “Cat Balou” and “Catobar Hornets” qualify, while the group named “catastrophy” does not (case mismatch for “C”). **Overrides any ‘coalition’ and ‘type’ attribute**; when you add a ‘group’ attribute, those other attributes are ignored. Defaults to <no group restriction>

Name	Description
	NOTE: requires module cfxMX to load before radioMenus
type types	Restricts access to this menu to player units who control a unit that matches one of the listed types. You can list multiple types, separated by a comma (e.g., “F-15C, A-10A”) Supports the class-wildcards ‘plane’ (all fixed-wing players) and ‘helo’ (all rotor-wings controlled by players). When you also supply a ‘coalition’ attribute, access to this menu is restricted to those players who match both. Defaults to <no type restriction> **NOTE:** requires module cfxMX to load before radioMenus
itemA	Name of itemA in this menu. If this attribute is omitted, no menu item appears
itemB	Name of itemB in this menu. If this attribute is omitted, no menu item appears
itemC	Name of itemC in this menu. If this attribute is omitted, no menu item appears
itemD	Name of itemD in this menu. If this attribute is omitted, no menu item appears
A!	DML flag to bang when itemA is chosen. Defaults to <none>
B!	DML flag to bang when itemB is chosen. Defaults to <none>
C!	DML flag to bang when itemC is chosen. Defaults to <none>
D!	DML flag to bang when itemD is chosen. Defaults to <none>
cooldownA	Cooldown (in seconds) after itemA is chosen before it becomes available again. Defaults to 0 (immediately available again)
cooldownB	Cooldown (in seconds) after itemA is chosen before it becomes available again. Defaults to 0 (immediately available again)
cooldownC	Cooldown (in seconds) after itemA is chosen before it becomes available again. Defaults to 0 (immediately available again)
cooldownD	Cooldown (in seconds) after itemA is chosen before it becomes available again. Defaults to 0 (immediately available again)
busyA	Message to display when itemA is chosen while cooldown is still active. Defaults to "Please stand by (<s> seconds)". Supports Time Wildcards <s>, <m>, <h>, <:s>, <:m> and <:h>
busyB	Message to display when itemB is chosen while cooldown is still active. Defaults to "Please stand by (<s> seconds)". Supports Time Wildcards <s>, <m>, <h>, <:s>, <:m> and <:h>
busyC	Message to display when itemC is chosen while cooldown is still active. Defaults to "Please stand by (<s> seconds)". Supports Time Wildcards <s>, <m>, <h>, <:s>, <:m> and <:h>
busyD	Message to display when itemD is chosen while cooldown is still active. Defaults to "Please stand by (<s> seconds)". Supports Time Wildcards <s>, <m>, <h>, <:s>, <:m> and <:h>
valA	Overrides radioMethod (see below) when this item is chosen to set flag A! according to this method. Example: #3 Defaults to <none> (no override)
valB	Overrides radioMethod (see below) when this item is chosen to set flag B! according to this method.

Name	Description
	Example: #3 Defaults to <none> (no override)
valC	Overrides radioMethod (see below) when this item is chosen to set flag C! according to this method. Example: #3 Defaults to <none> (no override)
valD	Overrides radioMethod (see below) when this item is chosen to set flag D! according to this method. Example: #3 Defaults to <none> (no override)
ackA	Acknowledge message when itemA was selected. Broadcast to all/coalition when itemA was chosen and not on cooldown. Supports wildcards. Defaults to <none> - no acknowledging message
ackB	Acknowledge message when itemB was selected. Broadcast to all/coalition when itemB was chosen and not on cooldown. Supports wildcards. Defaults to <none> - no acknowledging message
ackC	Acknowledge message when itemC was selected. Broadcast to all/coalition when itemC was chosen and not on cooldown. Supports wildcards. Defaults to <none> - no acknowledging message
ackD	Acknowledge message when itemD was selected. Broadcast to all/coalition when itemD was chosen and not on cooldown. Supports wildcards. Defaults to <none> - no acknowledging message
ackASnd	Name of sound file to play as a response to choosing itemA. Broadcast to all members of that group or everyone Default to <none>
ackBSnd	Name of sound file to play as a response to choosing itemB. Broadcast to all members of that group or everyone Default to <none>
ackCSnd	Name of sound file to play as a response to choosing itemC. Broadcast to all members of that group or everyone Default to <none>
ackDSnd	Name of sound file to play as a response to choosing itemD. Broadcast to all members of that group or everyone Default to <none>
method radioMethod	DML method to bang flags. Defaults to ‘inc’, meaning that each time that a menu item is chosen, the flag’s number is increased, generating a signal. To emulate ME’s native menu method that sets a flag to a value <v>, use that number <v> as method
radioTriggerMethod	Watchflag method for inputs Defaults to “change”
removeMenu?	Watchflag that triggers removal of entire menu
addMenu?	Watchflag that triggers adding the menu if it wasn’t shown or removed previously
menuVisible	When set (as per default) the menu is shown at the start of the mission. When set to false, the mission starts up with the menu hidden and requires a signal on addMenu? to appear Default to true (menu is visible on mission start)

Name	Description
radioMainMenu	Name of the menu to install in the Communications→F10 Other menu or the *radioMainMenu* that you refer to in *attachTo:* (see below). **MANDATORY**
attachTo:	By default, a radioMainMenu is added to a mission's 'F10: Other' menu. You can use radioMainMenu zones to cascade radioMainMenu or group radioMenus. If you use "attachTo:" (mind the colon ":"), this radio main menu attaches to the radioMainMenu that is described as value for this attribute. The trigger zone that you refer to as value must exist, and have a *radioMainMenu* attribute. Defaults to <none> (this radioMainMenu is installed into F10-Other)
coalition	The coalition that has access to this main menu. Applies to all attached menus. If omitted or set to 'neutral', *all* coalitions have access. 'blue' or 'red' restricts access to that coalition. Defaults to <no coalition>
group groups	Restricts this menu and all attached menus to groups with that name. Supports comma separated groups names, e.g. "Eagles 5, Uzi One, Aleph" will make this menu available to all members of those groups. **Supports wildcard** names: If you supply a group name "Cat*", all groups **that start with** "Cat" (WARNING: CASE SENSITIVE!) get access to this menu. For example, groups name "Cat Balou" and "Catobar Hornets" qualify, while the group named "catastrophy" does not (case mismatch for "C"). **Overrides any 'coalition' and 'type' attribute**; when you add a 'group' attribute, those other attributes are ignored. Defaults to <no group restriction> **NOTE:** requires module cfxMX to load before radioMenus
type types	Restricts access to this main menu (and all menus attached to this main menu) to player units who control a unit that matches one of the listed types. You can list multiple types, separated by a comma (e.g., "F-15C, A-10A") Supports class-wildcards '*plane*' (all fixed-wing players) and '*helo*' (all rotor-wings controlled by players). When you also supply a 'coalition' attribute, access to this menu is restricted to those players who match *both*. Defaults to <no type restriction> **NOTE:** requires module cfxMX to load before radioMenus

Cyclic references in mainMenu: Occasionally, people get strange ideas. One of them is to try and create a menu structure that points to itself, which creates a loop. radioMenus has code

built in to detect this, complain, and will refuse to work with such a structure. It'll process the other menus, though.

5.3.28.5 Using the module

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

To use

- add the 'radioMenu' attribute with a value (e.g. "Reinforcements") to a zone, which creates a new menu in the Communications→F10 Other menu.
- Add other attributes as needed

5.3.29 delicates

5.3.29.1 Description

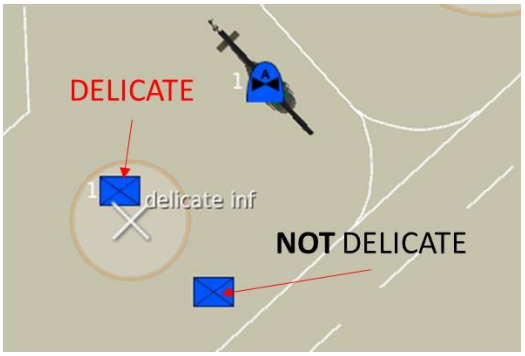
There comes a time where you want a unit/object/cargo to explode given the slightest provocation (think *"Le Salaire de la peur"*). This is the module that can do it for you: as soon as any unit designated as 'delicate' is damaged, it explodes.



DESIGNATING OBJECTS/UNITS AS DELICATES

To designate units or objects as 'delicate', you place them on the map in ME and a trigger zone around them zone with a 'delicates' attribute. All static objects and units inside that zone become 'delicate'.

Delicate status is determined on unit/object level, not group level, so it is possible to create a group in which only one unit is delicate.



Note that 'delicate' status is determined at mission start. Therefore, units that enter the 'delicates' zone later will not become delicate, and units that initially were inside the zone and later leave it remain delicate. Or, if you link a delicates zone to a unit, moving the delicates zone will not make those objects/units delicate that it encounters after mission start.

TRIGGERING A DELICATE OBJECT / SAFETY MARGIN

When a delicate object suffers damage, it explodes. This means that to trigger the explosion, you must first damage the unit/object. Note that this can be more difficult than you think: tanks, for example, do not get damaged when they are hit by machine gun fire, so firing at a tank with a machine gun will not trigger a 'delicate' tank.

You can set a safety margin to allow a small amount of damage to the object by adding a ‘safetyMargin’ attribute to the zone. The object only explodes if total damage exceeds that safety margin

Once a delicate object explodes, the zone can send out a signal on the delictaeHit! output.

EXPLOSION STRENGTH

When a delicate object is triggered, it explodes. The strength of that explosion is determined with the ‘power’ attribute



Remember that explosions can damage surrounding objects, so triggering one delicate object can trigger a chain reaction.

BLOWING IT ALL SKY HIGH

You can tell the zone to blow up all their delicates by sending it a signal on the blowAll? input. If you feed the delicateHit! output into this input, you essentially create a ‘one hit kills all’ scenario

LIMITATIONS

Delicate status does not work with cloners or scenery objects.

5.3.29.2 Dependencies

Delicates requires dcsCommon and cfxZones.

5.3.29.3 Module Configuration

To configure the delicates module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false

5.3.29.4 ME Attributes

Name	Description
delicates	Marks all static objects and units that are inside this zone when the mission starts up as ‘delicate’. If they get damaged at all, they immediately explode. Note that if these units move later outside this zone, they remain delicates, while units that move inside this zone later do NOT become delicates.

Name	Description
	Value of this attribute is ignored **MANDATORY**
power	Strength of the explosion when the delicate object is triggered. Defaults to 10
f! out! delicatesHit!	Flag to bang! when one of the units/objects defined by this zone explodes. Defaults to <none>
method delicatesMethod	DML Method for output flags Defaults to 'inc'
remove	When set to true, the delicate object/unit is removed from the game when it explodes. Defaults to true
triggerMethod delicateTriggerMethod	Watchflag method for all inputs
blowAll?	Sending a signal on this input causes all surviving delicates defined by this zone to blow up. When you feed back the delicatesHit! Signal into this input, one hit kills all.
safetyMargin	A number that defines how much damage the object can sustain (relative to its initial life) before it explodes. Expressed as a fraction from 0 (0%) to 1 (100%). For example, 0.1 means that the object can sustain 10% damage before it explodes. Defaults to 0 (any damage makes it blow up)

5.3.29.5 Using the module

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

To use

- add the 'delicates' attribute to a zone
- place objects/cargo/units inside that zone. These will be delicate during the mission
- Add other attributes as needed

5.3.30 Impostors

5.3.30.1 Description
Impostors allow you to replace AI-controlled units with identical-looking static objects on-the-fly, and turn them back into AI-controlled units when you need them. This can save a lot of CPU performance because every AI-controlled unit requires at least a small amount of CPU to control (look around, decide what to do next), while static objects do not.

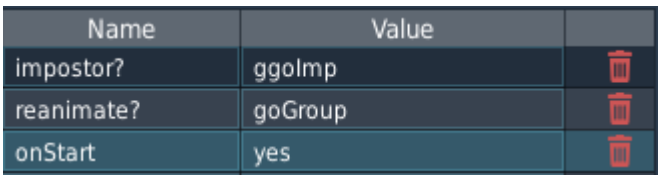
Use this for late/on-demand hot-starting aircraft and units/groups that are only required to be animated/controlled when players (i.e. people who see them) are close enough: airfield traffic etc. that you can safely 'switch off' when no one is around, or for late-activating aircraft that should be on their slots when the player is still at the airfield. In short, use Impostors to dial up the eye candy without tanking the mission's performance

You can also use Impostors to make SAM sites 'invisible' to RWR until they are activated – this has the added benefit that they are still visible to high-flying (stand-off weapons range) visual detection etc.

ABILITIES
Impostors are simply the static object (uncontrolled) version of a unit and therefore look exactly the same and use less CPU (they still use nearly the same amount of GPU to draw, though). DML's impostor zones

- Work for all groups (aircraft, ships and ground units)
- Preserve orders, path information etc.
- Can optionally (since it's a common use case) automatically turn AI-controlled units into static objects at mission start
- Allow you to switch between Impostor and AI-controlled units using flags
- Support multiple changes between impostor (static) and AI-controlled units
- Correctly handles destroyed units/objects when transitioning between Impostor/AI-controlled and back.
- Allows hot-start for aircraft when they reanimate (i.e. they start moving a second after changing to AI controlled units)
- Handles multiple groups per zone
- Integration with groupTracker
- Impostors support a zone's linkedUnit attribute when that zone links its movement to a unit that can become an impostor

HANDLING IMPOSTORS
Setting up groups to give them the ability to turn into static objects is easy: simply place a Trigger Zone with the 'impostor?' attribute. All groups that have at least one unit inside that zone become Impostor capable, i.e. can be turned into static objects upon command.



To turn all units of that group (that are still alive) into impostors, simply send a signal on the flag that you passed as value in the 'impostor?' attribute, in our example above the (global)

‘ggoImp’.flag. The units are now ‘frozen’: they no longer move, nor respond to enemy activity. Be advised that active enemy units can and will try to damage impostors.

Since it is such a common use case, you can add an ‘onStart’ attribute that when set to true, automatically turns the groups into impostors at mission start.

Once your mission determines that the impostors should revert to AI-controlled units, simply give a signal on the flag connected to the “reanimate?’ input.

When units reanimate, their AI is given the original orders from mission start (a DCS limitation), so it may cause some unexpected behavior with units that have move orders (they start moving towards the first waypoint – not the initial waypoint, the first waypoint)

Also note that Impostor behavior for aircraft can be somewhat unexpected due to the way DCS handles aircraft spawning.

- When airborne aircraft are turned into imposters, they spawn on the ground instead.
- When turning such an impostor back into an AI-controlled unit, they become airborne again
- Aircraft tend to revert to their initial spawn point and then head to their first way point when turned back to their AI-controlled state

Impostor abilities are determined at mission start, and any unit that has been designated as a possible impostor does not have to stay inside the impostor zone to retain their ability to be turned into an impostor nor to be turned back from impostor to AI-controlled unit.

HANDLING UNIT/OBJECT DEATH

Any unit (while AI controlled) or static object (while being an impostor) that is destroyed will not respawn when the group changes between Impostor/AI controlled units. This is important to remember since Impostors (being uncontrolled) do not defend themselves but will be attacked by enemy AI controlled units.

Note that units that are merely damaged will regain full health the next time they change to the other state (Impostor/AI controlled)

HANDLING WIPEOUTS

If all units/impostors that are managed by an impostor zone are destroyed, the impostor zone will take itself offline to conserve even more CPU. Obviously, sending signals to change between Impostor or AI controlled units will have no effect.

When all managed units/impostors are wiped out, the Impostor Zone can send a signal on the ‘allDead!’ output. This happens only once, the first time the module discovers that all units/impostors are dead during the health check (that happens ups times per second)

“BLINK” – AI PLANE REANIMATION OPTION

The built-in DCS logic for spawning aircraft checks if the slot that the aircraft should span on is occupied, and if so, automatically chooses an alternate, free slot that fits the aircraft size. This can

Name	Value	
impostor?	hgoImp	
blink	0.5	
reanimate?	goHawg	

lead to a conflict with an Impostor plane that is to become active (turn into an AI-controlled aircraft): the slot is currently occupied by a static object. This can only happen on Imposter to AI-control transition and is no issue when a unit transitions to a static object.

To resolve this, set a zone-individual short 'blink' interval (default is -1: no blinking) that causes the module to first removes the static object, allow for a brief period of emptiness so DCS can register that the slot is free, and then spawn the AI-controlled unit on that slot. Visually, the aircraft briefly 'blinks' out of existence only to return quickly. Use a short blink interval (usually 0.1 is enough), so the change is barely noticeable.

A blink interval is not necessary for ground units nor ships.

INTEGRATION WITH GROUPTRACKER
You can have impostor tracked with the groupTracker module. Be advised that each time a tracked group turns into impostors they are removed from the tracker (creating removeGroup events) and when turned back into AI-controlled units that creates addGroup events on that tracker.

INTEGRATION WITH LINKEDUNIT
One of DML's abilities is that it can link a zone to a unit's position by the "linkedUnit" attribute. Impostors respect such a link by inheriting the zone when a unit is turned into a static object, and they return link ownership to the AI-controlled unit when turned back.

LIMITATIONS
When you are using impostors in your mission, please hold the following limitations in mind:

- Impostors do not defend themselves but will be attacked by enemy AI. See: shooting ducks in a barrel.
- Obviously, Impostors don't move
- Any damage short of destruction will not carry over when a group changes from one state to the other. A half-dead infantry unit changes into a fully healed static unit and vice versa.
- While it's fine to change ships between impostor and AI controlled units, it's instant death for any unit stationed on that ship. The same is true for any static objects that you place on AI controlled ships and then turn into impostors.
- Player aircraft cannot be turned into impostors.
- Does **not** integrate with cloners nor spawners.

5.3.30.2 Dependencies
Impostors require dcsCommon, cfxMX and cfxZones.

5.3.30.3 Module Configuration
To configure the impostors module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "impostorsConfig" (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 health checks performed per second. Defaults to 1

5.3.30.4 ME Attributes

Name	Description
impostor?	Marks all groups that have at least one unit inside this zone as potential impostor, giving them the ability to change between static object and AI-controlled unit. The value of this attribute is a Watchflag that triggers transition of all surviving units from AI-controlled to static object. **MANDATORY**
reanimate?	Watchflag that triggers transition of all surviving static units to AI-controlled units. They immediately restart any waypoint actions or route orders (if given). Ground units will start moving their first (not initial) waypoint.
triggerMethod impostorTriggerMethod	Method that triggers inputs (DML Watchflags) Defaults to “change”
onStart	If the value of this attribute is true, all units are turned into imposters at the start of the mission. Default is false
blink	The value of this attribute specifies the brief interval (in seconds) between removing the impostor and spawning of the AI-controlled units. Only required for AI aircraft. A good value is 0.1 – 0.2; a value of zero or negative value means no blinking. Defaults to -1 (no blink interval)
trackWith:	Name of a zone with a groupTracker attached. All units are added to that tracker when they are AI-controlled and removed when they are turned into impostors. Reanimating them subsequently will again add them to the tracker etc.
allDead!	DML flag to bang! when all groups that are managed by this impostor zone have been destroyed
method impostorMethod	DML method for output Defaults to “inc”

5.3.30.5 Using the module

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

To use

- Place zone over at least one unit of the groups that you want to add the impostor ability and to be managed by this zone.
- add the ‘impostor?’ attribute to the zone
- Add other attributes as needed

5.3.31 persistence

5.3.31.1 Description

This module provides all persistence services for your mission and must be present whenever you want to support saving and loading your mission (in the somewhat limited context of DML). This module gives your mission access to “manual” save in addition to auto-save features, and saves the values of those flags that you specify in trigger zones via saveFlags attribute.

Please also refer to this document’s ‘Persistence’ section for more detailed information about DML’s persistence abilities and how to set up DCS to allow persistence.

Note:
This module provides DML’s persistence foundation as service to other modules. To persist specific ‘things’ that are outside of DML’s module “brick” scope (for example, units, warehouse inventories), you need a few other modules to provide the actual function (see unitPersistence or WHpersistence).

That being said, most DML modules automatically and transparently integrate with persistence, there is nothing for mission designers to enable or add, persistence should simply work.

5.3.31.2 Persistence Location (mission data)

You control where persistence saves your mission’s data with the “root”, “serverDir”, “saveDir” and “saveFileName” attributes. Assuming a mission named “coolMission.miz” running from a standard DCS Install, persistence saves data as follows:



In a fully defaulted configuration (i.e. out-of-the-box), persistence uses the following defaults:

attribute	Default
root	The directory that you configured DCS to be the ‘home’ directory. In a freshly DCS install that would usually be C:\Users\(user name)\Saved Games\DCS\ Defaults to what DCS tells persistence is the current home directory
serverDir	The directory name inside “root” that contains all missions. It’s usually called “Missions\” and that is what persistence defaults to
saveDir	A folder (allocated if it doesn’t exist) inside serverDir where persistence will save the mission data into a separate file. You can use this to pool multiple missions’ data into the same folder. Defaults to “<mission name> (Data)” (see note below) **IMPORTANT NOTE** If you completely omit persistence’s config zone, it reverts to simplified save mode, and defaults saveDir to “” (empty string), saving the mission’s data file directly into the serverDir.

attribute	Default
saveFileName	The name for the data (plain text in JSON format, can be edited with any text editor) file inside saveDir. Defaults to “<mission name> Data.txt”

5.3.31.3 Persistence Location (shared data)

Some modules (e.g. *Scribe*) support sharing their data with other missions. By default, these modules save their data inside the mission data file (see above). When enabling shared data, these modules read from and write data to a different file that is located at a different location on your storage, so that other missions can also access (read and write) that data.

Just like before, you can control where that data is written, and what name you give to that data file:



In the example above, the scribe module has a “sharedData” attribute that allows you to set the name of the data file that it looks for to load its data. All shared data files should reside in the same location (a recommendation only), and you can control where that file is going to be located.

attribute	Default
root	The directory that you configured DCS to be the ‘home’ directory. In a freshly DCS install that would usually be C:\Users\(user name)\Saved Games\DCS\ Defaults to what DCS tells persistence is the current home directory
serverDir	The directory name inside “root” that contains all missions. It’s usually called “Missions\” and that is what persistence defaults to. Server Administrators may want to change this to their needs.
(immutable)	The name of the directory in which shared data is placed. **This cannot be changed** and is always “DML-Shared-Data”.
??? (module-specific data sharing attribute name)	The name for the data (a plain text-file containing data in JSON format, can be edited with any text editor) file inside “DML-Shared-Data”. To share data between missions, all that a module must do is use the same value for the sharing attribute in different missions. Note that the name for the data sharing attribute is not controlled by the persistence module but the module that owns the data (e.g. *Scribe*) Multiple modules can use the same data file inside DML-Shared (e.g. the modules *Scribe* and *xyzUnknown*), so you do not have to create separate data files for each module inside shared data. If a module uses the same file name in shared data in multiple missions, these missions all access the same date file across multiple missions. *Example:* The *Scribe* module supports a “sharedData” attribute to set the name of the file used in the shared data folder. If you use the value “Angels”

attribute	Default
	for this attribute *in multiple missions* that use scribe, these *Scribe* module incarnations share their data between all missions that use "Angels" as value for this attribute, and the file "Angels.txt" inside "DML-Shared-Data" holds this shared data.

5.3.31.4 Redundancy
When you share data for a module, persistence also writes the same data to the mission's main data storage file for redundancy so that if the shared data file accidentally gets erased, the mission can fall back.

This means that if you share data between multiple modules and want to reset the data, you must erase the mission's and the shared data.

5.3.31.5 Dependencies
Persistence requires dcsCommon and cfxZones.

In order to save and load data, DCS must also be configured to read/write data. It gracefully handles situations where the module detects that it can't read or write data. The mission runs normally, it merely can't load mission data on start.

5.3.31.6 Module Configuration
You can configure the persistence module using a trigger zone in ME named "persistenceConfig" and add the following attributes

Name	Description
verbose	A value of "true" turns on debugging messages. Default is "false"
versionID	If present, this turns on version matching. When a mission starts up, persistence checks the value provided via the Zone with the one saved. If they do not match, the entire save data is discarded, and the mission starts fresh Defaults to <none>
root	Path to the DCS standard directory (usually "C:\userName\saved games\DCS.openbeta\" or "C:\userName\saved games\DCS\"). This value is passed from DCS to persistence. You can change this to adapt your missions to conform with more elaborate server setups. If you change this. Be sure that you know what you are doing, and initially have verbosity set to true, so you can see which directory your mission will save to. Defaults to your currently running DCS instance's write dir.
serverDir	Path from the root directory (see above) to the Missions directory. Use this if you set up your DCS different (usually important for dedicated servers). Defaults to "Missions\"
saveDir	Name for the mission's data directory. Defaults to "<mission name> (data)". This directory is created in the serverDir automatically if it does not exist

Name	Description
	If you set saveDir to “”, the mission saves its data directly into serverDir Defaults to “<mission name> (data)” if a configuration zone is present, none without configuration zone (i.e. the data is written into serverDir)
saveFileName	Name for the file that persistence uses to write mission data. Defaults to “<mission name> Data.txt”
saveInterval	Controls auto-save. Any value larger than zero will turn on auto save. The value you give here is the number of **minutes** between auto saves. Auto-saves co-operate with manual saves, so you can use both methods in your mission Defaults to `-1` (auto-save off)
saveNotification	When set to true, each time that the mission is persisted, a text notification is sent to all players. Default is true (notify players when saving)
cleanRestart?	DML Watchflag. A change signal on this input triggers a “fresh start” request: next time the mission starts up, it won’t load mission data. Defaults to <none>
saveMission?	DML Watchflag. A change signal on this input triggers a ‘manual’ (as opposed to automated) save. Defaults to <none>

5.3.31.7 ME Attributes

You can use Trigger Zones with the ‘saveFlags’ attribute to list the flags that persistence should save. Remember that you can add this flag to any trigger zone, especially zones with other modules.

Name	Description
saveFlags	A list of flags that you want to be saved with the mission. Supports local flags (e.g., “*go”) and numbered ranges (e.g. “3-17”). **MANDATORY**

Since **persistence loads before any other modules**, and you **can add the “saveFlags?” attribute to any Trigger Zone** (with other modules), you can **use this to set up module’s input flags to the correct value** to prevent/cause them to fire as these values are loaded into the default state when that module starts up later.

5.3.31.8 Using the module

See this manual’s ‘Persistence’ section

5.3.32 unitPersistence

5.3.32.1 Description
This module requires the “persistence” module to be added to the mission.

The unitPersistence module transparently saves the positions of all *ME-placed* AI units and static objects that are still alive at the point of save, and restores them when the mission starts. **Persisting dynamically spawned units is handled by the modules that spawn them** (e.g. cloners, spawn zones, owned zones etc.) so please check their description for persistence support.

Due to the way DCS works, there are the following quirks to observe when a mission loads from saved data:

- *All Units*
  The AI assumes it spawns a unit at its initial point (IP), and then proceed to waypoint 1 (WP1, if given). Although DML saves the unit’s progress, it can’t query the AI which waypoint it’s currently approaching and curtail the route accordingly. This means that if you have routes assigned to groups, any loaded group will start moving towards waypoint 1 after loading. If you design a mission with persistence, make sure that you take this into account: Make sure that a group has at most one waypoint (in addition to the initial waypoint where it spawns). If you add more waypoints, it helps if all additional waypoints are close together, so that if the units return to WP1

  Also remember that when groups load, any surviving unit loads at full health, with a full complement of munitions

- *Aircraft Groups (includes Helicopters)*
  If the initial waypoint is on the ground, the loaded aircraft will always start from the ground, no matter how far it had progressed previously. This may change later if a) DCS changes controller logic to allow access to current waypoint info or b) I get my smart route analysis code running that can curtail an AI’s route

- *Player Groups*
  Player groups aren’t persisted

- *Ships*
  Due to a quirk with carriers, ship groups are currently only persisted in a ‘fully dead’ way: if the entire naval group is destroyed, it won’t re-appear after load. If just one unit of a naval group is barely limping along, the entire groups re-spawns after load, at full health and fully stocked.
  Ship group’s positions currently aren’t persisted, they spawn at the ME initial position

- *Static Objects (incl. Cargo)*
  No restrictions, and destroyed statics respawn as destroyed with visual representation. Cargos respawn in a loaded mission as cargo at the location that they were moved to at the point of save

5.3.32.2 Dependencies
Requires dcsCommon, cfxZones, persistence and cfxMX.

5.3.32.3 Persistence
This module persist all non-dynamic ground units and static objects.

5.3.32.4 Module Configuration
None.

5.3.32.5 ME Attributes
None.

5.3.32.6 Using the module
Add unitPersistence to your mission. Smile.

5.3.33 sequencer

5.3.33.1 Description

Many things happen in a strict order, and with variable intervals between them. Since DCS has no specific conductor/director abilities to make sure things happen in the correct sequence, making sure that one thing happens after another can be tedious.

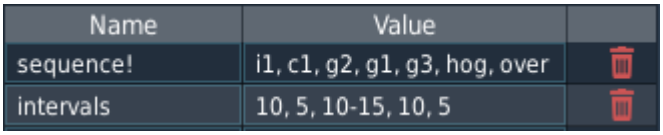
This is where the sequencer module comes it. Its sole purpose is to ensure that a sequence of flag changes, that, for example flag “moveRefueler” is changed before flag “startAirGroup" changes.

Sequencers have many uses in your mission

- Ensure correct staging of your mission (e.g. spawning enemies of the next stage when all enemies in the current stage are destroyed)
- Easily run conversations/narratives where you play sound files and/or messages in a sequence with pauses in-between
- Coordinate ‘cut-scene’ or similar behaviors of vehicles (move refueller away from aircraft, start aircraft, have another crew member arrive, have aircraft take off)
- Schedule interruptible tasks/events (e.g. re-starting mission, inbound enemies) with warnings in intervals they happen
- Easily schedule “time or event”-style stages

Each stage in a sequence has an easily controllable duration, and a sequencer can be paused, resumed and reset at any time

GENERAL BEHAVIOR



You supply a sequencer with a list of flags or ‘stages’, and when you start the sequencer, it will, one after the other send signals to the flags that you have listed, in exactly the order that you have listed them. For example, if you listed flags “A, B, D, Z”, the sequencer will signal these flags in the order A. B, D, Z.

Once it has reached the final flag (stage) in the sequence, it can either restart the entire sequence, or send a signal on the “done!” output and terminate the sequence.

INTERVAL: PROGRESSING THE SEQUENCE

There are multiple ways how the sequencer progresses from one stage to another. In general terms, after the sequencer signals a flag, it waits before it signals the next flag. So how long does it wait, and what does the sequencer wait for?

There are multiple ways to progress the sequence, and they work cooperatively

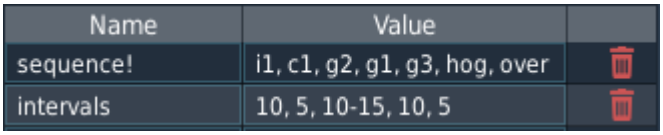
- A time limit
- A signal on the nextSeq? Input

Whatever comes first (time-out on the timer or signal on nextSeq?) will progress the sequence.

Time-driven progress
You can supply a list of durations with the “intervals” attribute, and the sequencer then uses these to set the time interval until the next sequence stage starts. The sequencer is highly flexible with regards to how it handles the duration between stages:

- A single value (e.g., “20”) results in the duration between all stages being that same duration (20 seconds in this case)

- If you supply a list of values (e.g. “10, 20”), the sequencer uses these one after the other for each sequential stage. When it reaches the end of the list of durations, it ‘rewinds’ and begins again with the first duration. This means that the number of durations does not have to match the number of flags in your sequence. For example, if your sequence is “A, B, C, D, E, F”, and your durations is “10, 20”, the resulting sequence is:



  A (10 sec) B (20 sec) C (10 sec) D (20 sec) E (10 sec) F

- Each duration in the list can be a range (e.g. “10-30”), in which case the sequencer picks a random value in that range (e.g., 23)

- If you do not give any duration, the sequencer uses a 24 hour wait period (meaning that stage progress is made entirely via the nextSeq? input.

Flag-Event driven progress
Any time that a sequencer is waiting for a time-out, a signal on nextSeq? ends that stage and progresses to the next one.

Combined Progress
Many missions use a schema “do this when the player takes off, or after at latest 15 minutes, whatever comes first”. A sequencer supports this out of the box. Feed the flag that signals a player’s departure into the ‘nextSeq’ input, and set the duration to 15*60 = 900 sec.

SEQUENCER CONTROL
Once a sequencer is started, it runs until it has run through the last stage (unless it is set to loop, in which case it runs through the sequence endlessly. At any time, a sequencer can be paused, which will also pause the timer until it is resumed.

Also at any time, a sequencer can be reset, which causes it to revert it’s initial setting (ready to run, started if onStart is true, first sequence, first duration)

Starting a Sequence
When a mission loads, the sequencer checks to see if the onStart attribute is true (defaults to false), and if not, settles into stopped mode. So to start a sequence, issue a signal on the 'startSeq?' input.

Pausing and Continuing
The 'startSeq?' and 'stopSeq?' inputs start and stop a sequence. If a sequence is running, and there are 10 seconds left on the current stage when you stop the sequence, it'll resume with 10 seconds left on the time when you issue the 'startSeq?' input.

Reset
When you give a sequencer a signal on the reset? Input, it reverts to the state that it started up in. This means that if 'onStart' is false, it needs a signal on 'startSeq?' to begin.

THE FIRST SEQUENCE: SIGNAL-WAIT VS WAIT-SIGNAL (zeroSequence)
So what constitutes the very fist stage? Should the sequencer immediately signal the first flag in the sequence when it starts and then wait, or wait the first duration and then signal? The attribute "zeroSequence" controls this. By default, this attribute is true: the sequence starts with a signal and then enters the first wait, it runs as a 'signal-wait' sequence.

If you set zeroSignal to false, the sequence starts with the first wait duration, and at the end of the first wait issues the signal on the first flag – a wait-signal sequence.

Remember this slight difference when you calculate your durations.

5.3.33.2 Dependencies
Sequencer requires dcsCommon, cfxZones.

5.3.33.3 Module Configuration
To configure the sequencer module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false

5.3.33.4 ME Attributes

Name	Description
sequence!	A comma-separated list of flags that should be banged! in exactly that order. Example: "startGround, startSAM, startSirens" Value of this attribute is ignored

Name	Description
	MANDATORY
interval invervals	A comma-separated list of durations (in seconds) between the flag banging. The number of intervals does not have to match the number of flags given under sequence!, the values simply repeat. A single value means that the same interval is applied between all stages. You can supply value ranges instead of a single value, and the sequencer picks a random value from that range. Example: “10, 12-17, 33” Defaults to 86400 (24 hours)
next?	A signal on this input causes a running sequencer to end the current count-down to the next stage, and immediately start the next stage. Defaults to <none>
onStart	If set to true, the sequencer starts the first sequence 0.25 seconds into the mission (this delay allows all other modules to load and initialize before the first signal is sent from the sequence) Defaults to false (sequencer requires a startSeq? signal.
zeroSequence	Controls if the sequencer operates in “signal-wait” or “wait-signal” mode: if it starts with a signal (zeroSequence is true) or a wait (zeroSequence is false) Defaults to true (start with signal, “signal-wait” mode)
loop	If true, the sequence repeats endlessly or until stopped with ‘stopSeq?’ or ‘reset’. If false, the sequence ends after the last stage, and a signal is sent on done! Defaults to false
done! seqDone!	The signal to send when the sequencer has ended the sequence Defaults to <none>
startSeq?	A signal on this input starts a paused or un-started (when onStart is false) sequencer. If the sequence was paused, the count-down resumes at the moment it was paused. If the sequence is already running it has no effect. Note that a sequence that has ended (run through all stages) cannot be restarted with startSeq? but must be reset first. Defaults to <none>
stopSeq?	A signal on this input pauses a running sequence. The current state of the stage’s timer is preserved. If the sequencer is already paused, this has no effect. Defaults to <none>
resetSeq?	A signal on this input resets the sequencer to the initial state: paused (if onStart is false), sequence stage one, duration one. A sequence that has ended (run out of stages) can be reset and then started again. Defaults to <none>
method seqMethod	The method to use for all outputs. Defaults to ‘inc’
triggerMethod seqTriggerMethod	The trigger method for all inputs. Defaults to “change”

5.3.33.5 Persistence
Sequencers support persistence. When you load a mission, any running sequencer resumes at the stage and remaining time for that stage from the point where they were saved.

5.3.33.6 Using the module
To use sequencers in your mission, add the script to the mission as a DOSCRIPT action during Mission Start

To use

- add the ‘sequence!’ attribute to a zone and add the flags you want to bang! in sequence
- Add other attributes as needed

5.3.34 fireFX

5.3.34.1 Description

This module is similar to ‘smoke zone’, except that it creates a highly visible fire/black smoke visual effect that can be turned off much quicker and that can be controlled in size and visual intensity.

Note that fire/smoke effects are purely visual, they do not damage vehicles or infantry standing inside them, nor aircraft flying through them. They also do not produce any thermals or wind effect.

SIZE MATTERS

DCS provides the same effect in different sizes, and you can set the fire effect’s size with the module’s main ‘smokeFX’ keyword to the following:

- “small” or “S”
- “medium” or “M”
- “large” or “L”
- “huge” or “H” or “XL”

ABOVE GROUND LEVEL?

Sometimes (especially in maps that have oil rigs) you need the flame to be above ground, usually at the top of a large chimney. You can do that by adding an AGL attribute, which sets the height above ground in meters.

NO SMOKE WITHOUT FIRE? YES THERE IS!

The effect can be a fire/black smoke combo, or you can have just some black smoke. There is no setting that allows flames without billowing smoke.

DENSITY

You can control the ‘thickness’ of the smoke, making it appear near-translucent (like with a very hot fire), or near black, like from an oil fire.

FLAME ON!

Most importantly, you can control when fire/smoke should appear and disappear. The two inputs start? and stop? allow you to turn the effect on and off at will (in sharp contrast to the smoke zone effect, that you can only turn on, and then have to wait for some minutes until it fades out).

Also, you can opt to have the fire/smoke effect visible from the mission’s beginning by using the ‘onStart’ attribute.

IT TAKES A LITTLE BIT OF TIME

When you tell fireFX to stop the effect, DCS now (as of version 2.9?) takes a couple of

seconds to slowly fade out the flames. This is a nice addition, even if sometimes unexpected and makes the effect a bit uneven (fires start near-instantaneous).

MORE THAN ONE
You can tell fireFX to start more than one fires inside the zone via the 'num' limit. If you stell fireFX to start more than one fires in the zone, their locations inside the zone is randomized.

RANDOMIZED
You can have the fire(s)'s location inside the zone randomized by using the appropriate attribute.

5.3.34.2 Dependencies
fireFx requires dcsCommon, cfxZones.

5.3.34.3 Module Configuration
To configure the fireFX module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false

5.3.34.4 ME Attributes

Name	Description
fireFX	Tells DML that you want fire/smoke effects inside the zone. The value of this field specifies the size of the effect itself. Currently DCS recognizes the following: • "small" or "S" • "medium" or "M" • "large" or "L" • "huge" or "H" or "XL" Smoke is always colored black. Flames are optional (see the 'flames' attribute) Note that *this effect is visual only*. Vehicles or troops inside the fire aren't damaged by the fire at all. Defaults to "small" (if none of the given values is recognized) **MANDATORY**

Name	Description
agl	Altitude **in meters** above the ground where the effect should start. Often used with oil rigs or chimneys to create torch/smoke effects at their top. Defaults to 0 (directly on the ground)
flames	If you supply “false” for this attribute, only smoke (no fire) is displayed. Defaults to true (flames and smoke)
density	“Thickness” or visibility” of the smoke produced by this effect.
start?	DML watchflag (input) for when the effect should start. Defaults to <none>
stop?	DML watchflag (input) for then the effect should stop **Note:** unlike the smoke zone module, which can take several minutes for the smoke to stop, this fireFX’s smoke and flames take roughly 10 seconds to peter out) Defaults to <none>
onStart	If set to true, the mission starts with the effect on Defaults to false (no effect on start)
triggerMethod fxTriggerMethod	DML method for inputs Defaults to change
num	The number (e.g. “3”) or range (e.g. “2-7”) of fires to start. If you supply a range, each time that you start fires in the zone, a random number from lower to upper range (inclusive) of fires is started. Defaults to 1 (one fire)
rndLoc	If true, the location(s) of the fire(s) is randomized. If you are allowing more than 1 fires to be started, this value is set to true. Defaults to pause (fire starts at center)

Note: if onStart is false and no start? Flag is given, that zone will provide a warning at start-up that it’s effect can’t be started.

5.3.34.5 Using the module

To use fireFX in your mission, add the script to the mission as a DOSCRIPT action during Mission Start

To use

- add the ‘fireFX’ attribute to a zone and add any attributes to control appearance and control

5.3.35 LZ (Landing Zone)

5.3.35.1 Description

For most DCS missions, it makes sense to synchronize actions to the point in time where a key aircraft takes off (to start the mission phase, and perhaps remove AI-intensive ground animations), or lands (to conclude, or kick of the next mission phase).

LZ gives mission designers the ability to create output signals whenever aircraft (fixed-wing and helicopters) take off or land.

This module can notify you (or other modules) when aircraft land or take off inside its zone. You have flexible control over which aircraft generate a landing or take-off event.

BASIC OPERATION

An LZ has two outputs: 'landed!' and 'departed!' that create a signal when any aircraft that matches some requirements lands or takes off inside the zone. By default, all aircraft in the game trigger these signals, and using attributes, you can narrow the range down to individual planes.



Note that for most fixed-wings, landing inside a zone usually means landing at, or taking off from, an airfield or carrier. LZ aren't limited to airfields and can be used, for example to detect landings on a dirt strip or make-shift airfield (grass in WW II).

And, of course, Helicopters and other VTOL aircraft can land and take off almost anywhere, where an LZ comes in handy to detect such an event.

WHICH AIRCRAFT?

Often, you are only interested to be notified when a specific aircraft or group takes off or lands, and LZ gives you attributes with which you can narrow the search by unit name, group name, type, and/or coalition.

- unit/group
  when given either attribute, you can list the names for units or groups (separated by comma), and if you use an asterisk wildcard '*' as the last letter in a name, all units or groups that match everything before the asterisk will be considered: if you are looking for "He*", "He", "Helium", "Heinkel" and "Hello-1-1" will all match.

- type
  type allows you to list the plane types (e.g., "A-10A") that you are interest in, separated by comma.

  LZ supports the following special types:
  - "ANY" – any plane.

- “HELO” – all helicopters
- “PLANE” – all fixed-wing aircraft

MODIFIERS
In addition to above selection criteria, LZ supports two modifiers to further narrow the search:

- coalition
  When given, all units that match the requested type are also checked against their coalition.

- playerOnly
  When set to true, all AI units are excluded

Note that it is perfectly legal to specify a LZ that, for example, detects only players from the red side by setting the LZ to “coalition=red” and “playerOnly=yes” without any unit, group, nor type attribute

THE ZONE IS HOT!
You can turn an LZ “on” and “off” via its “pause?” and “continue?” inputs. A paused LZ will no longer detect any landings or take-offs inside until it is told to continue.

5.3.35.2 Dependencies
LZ requires dcsCommon, cfxZones.

5.3.35.3 Module Configuration
To configure the LZ module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false

5.3.35.4 ME Attributes

Name	Description
LZ	Tells DML that this is an LZ that detects landing and take-off of aircraft inside it. The value of this attribute is ignored **MANDATORY**
landed!	Flags to bang! when a plane matching the criteria lands inside the zone Defaults to <none>
departed!	Flags to bang! when a plane matching the criteria lands inside the zone

Name	Description
	Defaults to <none>
coalition	When given, the coalition that aircraft must match. Supported are “red”, 1, “blue”, 2, “neutral” 0 When you specify “neutral” or 0, all coalitions match Defaults to 0 (any coalition)
playerOnly	When set to true, all AI planes are ignored Defaults to false (all players and AI are considered)
unit units	A comma-separated list of all unit names (case **in**sensitive) that should be considered. Supports wildcard “*” for the last character, in which case all groups that match that everything before the asterisk are considered: “Hog-*” will match “HOG-”, “hog-1-1” and “HoG-5-2”, but not “Hogger” When matching units, any ‘coalition’ setting is ignored Defaults to <none>
group groups	A comma-separated list of all group names (case **in**sensitive) that should be considered. Supports wildcard “*” for the last character, in which case all groups that match that everything before the asterisk are considered: “He*” will match “He”, “HELLO” and “heinkel-1-1”, but not “Hans Heinkel” Defaults to <none>
type types	List of types, separated by comma, that a unit must match at least one of (e.g. “A-10A”). Types must match exactly Additionally, the following special types are also supported • “ALL” or “ANY” – all aircraft match • “HELO” – all helicopters match • “PLANE” – all fixed-wing planes match Defaults to “ALL”
isPaused	When set to true, the LZ is paused at mission start Defaults to false (LZ is active on start)
pause?	DML Watchflag to pause the LZ Defaults to <none>
continue?	DML Watchflag to continue a paused LZ Defaults to <none>
method outputMethod	Method to the outputs Defaults to ‘inc’
triggerMethod lzTriggerMethod	Method that triggers inputs. Defaults to “change”

5.3.35.5 Using the module

To use LZ in your mission, add the script to the mission as a DOSCRIPT action during Mission Start

To use

- add the ‘LZ’ attribute to a zone and add any attributes to control detection of landings and take-offs

5.3.36 counter

5.3.36.1 Description

A very simple module that counts the number of times that it is triggered. Note that this can be emulated solely with flags, but using a counter module is more natural and easier for many designs. Due to DML's trigger method ability, counters can be easily used to detect certain conditions like a flag holding a certain value, and be used to count occurrences of that event.

BASIC OPERATION

Quite frankly, counters only have basic operation: set up the trigger condition (usually a change on the input flag), and then increase (of otherwise changes) the output flag. It's their simplicity that makes counters so attractive. Most of the magic comes from DML-inherent abilities: define a clever trigger condition and/or a good output method, and a counter can achieve seemingly fantastically complex results.

5.3.36.2 Dependencies

Counter requires dcsCommon, cfxZones.

5.3.36.3 Module Configuration

To configure the counter module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false

5.3.36.4 ME Attributes

Name	Description
count?	Tells DML that this is a counter that is triggered by this input **MANDATORY**
triggerMethod triggerCountMethod	Watchflag – the condition that triggers the input. Defaults to "change"
method countMethod	Output method. Defaults to "+1"
out! countOut!	The output to change when the counter is triggered. Defaults to <none>

5.3.36.5 Using the module

To use counter in your mission, add the script to the mission as a DOSCRIPT action during Mission Start

To use

- add the ‘count?’ and ‘countOut!’ attributes to a zone. Usually, you’d also supply at least one of the methods to fine-tune either what the counter counts (input), or how it counts (output)

5.3.37 valet (greet and goodbye players)

5.3.37.1 Description



Valet is a module that helps you to easily generate events, text message or sound effects for players who enter or leave a zone like airfields, Hospital zones, FARPs, naval groups, destinations etc. Valet has strong built-in abilities to assemble complex text messages and can send these or sound to that player unit. Although similar functionality can be assembled in DML when you put together some messengers, unit zones, and some changer modules, it makes sense to provide an advanced, dedicated module for this: it's such a common task for many missions.

Use Valet zones for airfields, FARPS, or any other cases when you want to address player-controlled units should they enter or leave a valet zone.

Basic Behavior

Name	Value	
valet		
greeting	<u>, contact Batumi center	
goodbye	<u>, frequency change app	

Valets react whenever player units enter or leave the area. Should a player-controlled unit enter the valet zone, greeting messages and sound files can be played to that unit. And should the unit leave the zone, different good-bye messages or sounds can be sent. Also, to avoid a staccato of messages when a unit skirts a valet's zone's border, their triggers have a built-in hysteresis effect: once a unit triggers a in/out transition, some time must elapse before it can trigger the opposite transition.

Greeting and Goodbye Messages

Valet zones support different text messages and sound files for both greetings (when a player unit moves into a valet zone) and goodbyes (when a player unit moves outside of a valet zone). Text messages are set with the "greet" and "goodbye" attribute, while sound files are set with the "inSoundFile" (played when the player enters the valet zone) and "outSoundFile" attribute. If you omit a greeting or goodbye attribute, or the text is empty, no text is displayed. Sound and text function independently from each other, so you can have a valet play an inSound (greeting sound), but display no text.

First time and Spawn greetings

Valets zones provide an optional special '*firstGreeting'* and "*firstInSoundFile*" attribute. If you supply such an attribute, that text or sound is played the first time that a player's unit enters the valet zone. On subsequent entries into the zone, the player will be greeted with the normal "*greeting*" and "*inSoundFile*" greetings. Note that you can use this feature to greet a player only once: simply provide only *firstGreeting* and no "greeting" attribute.

Many valet zones contain player spawn points. Since you may not want the valet to greet players that spawn in, you can suppress greetings to spawning units by setting the "greetSpawns" attribute to false.

(Re-)spawning Players
Valet Zones remember a player until they re-spawn. Should a player (re-)spawn into the mission, even if they respawn into the same unit (say, after a particularly enthusiastic landing), *all* valets instantly forget the player. If they then enter a valet zone, their entry is treated as a first entry into the zone (i.e., they are eligible for the first greeting / first sound effect).

The bouncer: Who will be greeted / farewelled
By default, a valet zone greets *all* player units equally. You can control to whom a valet responds with the following attributes:

- *coalition*
  only players of that coalition will be handled

- *types*
  only players who are controlling a unit that matches at least one of the listed types will be handled. Supports lists with wildcards (asterisk "*" at the end)

- *groups*
  only players who are part of one of the groups listed are handled. Supports lists with wildcards (asterisk "*" at the end)

- *units*
  only players who are controlling a unit that matches a name on the list will be handled. Supports lists with wildcards (asterisk "*" at the end)

Arrival and departure notifications
In addition to messages and sound effects, valet zones can also bang! on output flags whenever it triggers (i.e., an eligible player unit enters ort leaves the valet zone). Note that unlike the messages and sound effects, the outputs do not differentiate by player nor first greeting, they always trigger when the in/out conditions are met.

Greet and Goodbye message wildcards
The text messages that valet sends to units **supports wildcards like the messenger module** (see that section for a more elaborate description of message wildcards), so you can design the greeting and farewell messages to be dynamic and appear personal.

The following wildcards are supported for *greeting*, *firstGreeting* and *goodbye*:

- `<player>` - callsign of the player
- `<unit>` - name of the unit that the player is controlling
- `<type>` - type (e.g. "A-10A") of the unit that the player is controlling
- `<group>` - name of the group that the player's unit is in
- `<in>` - number of greetings that the player has received from this valet

- `<out>` - number of goodbyes that the player has received from this valet

- `<n>` creates a line feed
- `<z>` - valet zone's name as set with ME
- `<t>` - current mission time in the format "HH:MM:SS", e.g. "08:42:11". Format be further configured with the *timeFormat* attribute.
- `<lat>` - latitude of the valet zone's current position
- `<lon>` - longitude of the valet zone's current position
- `<ele>` - elevation of the valet zone's current position. Format be further configured with the *imperialUnits* attribute.
- `<mgrs>` - valet zone's current position in MGRS coordinates

- `<v: flagName>` - value of flag <flagName>
- `<t: flagName>` - value of flag <flagName> interpreted as time. Format be further configured with the *timeFormat* attribute.
- `<lat: unit/zone>` - latitude of unit/zone with that name
- `<lon: unit/zone>` - longitude of unit/zone with that name
- `<ele: unit/zone>` - elevation of unit/zone with that name. Format be further configured with the *imperialUnits* attribute.
- `<mgrs: unit/zone>` - mgrs of unit/zone with that name
- `<coa: flag/unit/zone>` - faction (e.g., "RED") of zone/unit, or the flag value interpreted as such.
- `<vel: unit/zone>` - velocity of unit/zone with that name. Format be further configured with the *imperialUnits* attribute.
- `<alt: unit/zone>` - altitude of unit/zone with that name. Format be further configured with the *imperialUnits* attribute.
- `<hdg: unit/zone>` - heading of unit/zone with that name
- `<type: unit>` - type (e.g. "A-10A") of unit with that name
- `<player: unit>` - player callsign who controls a unit with that name
- `<twn: unit/zone>` - (requires twn module): name of nearest map town/village
- `<loc: unit/zone>` - (requires twn module): location relative to nearest map town/village (e.g. "1.2km SW of Zugdid")

The valet zone's attributes '*valetTimeFormat'* and '*imperialUnits'* can (when given) control how time, velocity, and altitude are calculated/formatted. See the 'messenger' module for details.

5.3.37.2 Dependencies
valet requires dcsCommon and cfxZones to run

5.3.37.3 Module Configuration
To customize valet's global behavior,

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

Name	Description
verbose	Show debugging information. Default is false

5.3.37.4 ME Attributes

Name	Description
valet	Tells DML that this trigger zone is a valet **MANDATORY**
greeting	Message to be displayed if a player unit triggers the valet. The message can be heavily customized with dynamic wildcards. Valet supports the following wildcards in the greeting, which are replaced with current values as the mission runs: • `<player>` - callsign of the player • `<unit>` - name of the unit that the player is controlling • `<type>` - type (e.g. “A-10A”) of the unit that the player is controlling • `<group>` - name of the group that the player’s unit is in • `<in>` - number of greetings that the player has received from this valet • `<out>` - number of goodbyes that the player has received from this valet • `<n>` creates a line feed • `<z>` - valet zone’s name as set with ME • `<t>` - current mission time in the format “HH:MM:SS”, e.g. “08:42:11”. Format be further configured with the *timeFormat* attribute. • `<lat>` - latitude of the valet zone’s current position • `<lon>` - longitude of the valet zone’s current position • `<ele>` - elevation of the valet zone’s current position. Format be further configured with the *imperialUnits* attribute. • `<mgrs>` - valet zone’s current position in MGRS coordinates • `<v: flagName>` - value of flag <flagName> • `<t: flagName>` - value of flag <flagName> interpreted as time. Format be further configured with the *timeFormat* attribute. • `<lat: unit/zone>` - latitude of unit/zone with that name. • `<lon: unit/zone>` - longitude of unit/zone with that name. • `<ele: unit/zone>` - elevation of unit/zone with that name. Format be further configured with the *imperialUnits* attribute. • `<twn: unit/zone>` outputs the name of the nearest named location on the map. Requires the twn module • `<loc: unit/zone>` outputs position relative to nearest named map location. Requires twn module • `<mgrs: unit/zone>` - mgrs of unit/zone with that name • `<coa: flag/unit/zone>` - faction (e.g., “RED”) of

Name	Description
	zone/unit, or the flag value interpreted as such. • <vel: unit/zone> - velocity of unit/zone with that name. Format be further configured with the *imperialUnits* attribute. • <alt: unit/zone> - altitude of unit/zone with that name. Format be further configured with the *imperialUnits* attribute. • <hdg: unit/zone> - heading of unit/zone with that name. • <type: unit> - type (e.g. "A-10A") of unit with that name • <player: unit> - player callsign who controls a unit with that name. If the message text is empty, no message is displayed. Defaults to <empty message>, i.e., no greeting diosplayed
firstGreeting	When present, the text of this message replaces the message that is displayed when a player enters the valet for the first time since they (re)spawned. If the attribute is present but the text is empty, no message is displayed (i.e., this can be used to suppress the first greeting to a player) Supports all wildcards that *greeting* does. Defaults to <none, no separate first greeting)
goodbye	Message to display when a player leaves the valet zone Supports all wildcards that *greeting* does. Defaults to <empty>, i.e. no text message.
imperial imperialUnits	When set to true, wildcards return units for velocities, length etc. in "imperial" funny units (knots, feet, tablespoon etc.) Defaults to false (SI units are used: m, km/h)
timeFormat	The time format in which time values are displayed. Defaults to <:h>:<:m>:<:s>
duration	Time duration (in seconds) that a text message is displayed for Defaults to 30
inSoundFile	Name of the sound file that is to be played when a player enters the valet zone Defaults to <none>
firstInSoundFile	When present, this audio is played when a player enters the valet for the first time since they (re)spawned. Defaults to <none>, i.e., insoundFile is played
outSoundFile	Name of the sound file that is to be played when a player leaves the valet zone Defaults to the value of *inSoundFile*
greetSpawns	Controls if a unit that spawns inside a valet zone should be greeted. If set to false, a unit that spawns within a valet zone will not receive a greeting, and must exit and then re-enter the valet zone for its first greeting. Default is false (spawning units inside a valet zone will not be greeted but have to leave zone first)
method valetMethod	DML method for outputs Defaults to 'inc'
hi!	Output flag to set when an eligible player unit enters the valet zone

Name	Description
bye!	Output flag to set when an eligible player unit leaves the valet zone
coalition valetCoalition	When given, restricts the valet to players of that coalition (e.g., 'red'). Players of other coalition are ignored. Defaults to <none> - all players are eligible
types valetTypes	A list of unit types (e.g., "A-10A"), separated by comma. The valet zone is restricted to players who control units of that type. List supports end-wildcards, i.e., when the type ends on an asterisk "*", all types that match everything before the asterisk are matches. Example: "A-10*" matches "A-10A", "A-10C" and "A-10C_2", but not "AV8BNA". Types are not case sensitive. Defaults to <none>, all types permitted
groups valetGroups	A list of group names, separated by comma. The valet zone is restricted to players who control units who are members of the groups on the list. List supports end-wildcards, i.e., when the type ends on an asterisk "*", all group names that match everything before the asterisk are matches. Example: "atta*" matches "Attackers-1" and "atta99" but not "avenger-1" Group names are NOT case sensitive. Defaults to <none> - no group restrictions
units valetUnits	A list of unit names, separated by comma. The valet zone is restricted to players who control units with names on the list. List supports end-wildcards, i.e., when the type ends on an asterisk "*", all unit names that match everything before the asterisk are matches. Example: "atta*" matches "Attackers-1" and "atta99" but not "avenger-1" Unit names are NOT case sensitive. Defaults to <none> - no unit restrictions

S

5.3.37.5 Using the module

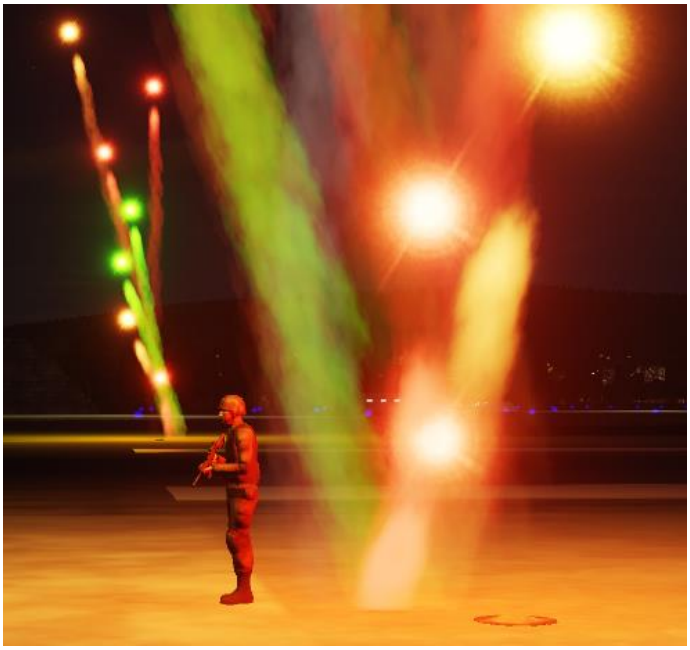
To use valets in your mission, add the script to the mission as a DOSCRIPT action during Mission Start

To use

- add the 'valet' attribute to a zone. Usually, you'd then also supply at least one of the attributes to define-behavior for *greeting* and *goodbye*.

5.3.38 flareZone

5.3.38.1 Description



Like *FireFX* and *SmokeZone*, FlareZone is a small, convenient module to perform a single function well. FlareZones launch flares from their zone's center. They are a lot easier to use than vanilla ME, and they offer significant Quality-of-Life enhancements over pure ME: random flare color, direction, duration and multiple flare launches. Also, FlareZones support salvos of flares with a randomizable duration for the salvo, all which can be helpful if you want to create some fireworks display.

Color

You can either choose one of the standard DCS flare colors (red, green, white, yellow).

Launch Direction

By default, all flares are launched towards North. You can change this by giving a compass heading in degrees (e.g., 270).

Salvo Size and Duration

When triggered, a single flare is launched immediately. You can change this so that multiple flares are launched over a user-selectable duration

Randomization

FlareZone supports the following randomizations:

- *Color*
- *Salvo Size:* minimal and maximal number of flares to launch
- *Duration:* minimal and maximal duration in seconds
- *Direction:* lower and upper bounds of heading to launch the flare. Only positive headings are supported.

5.3.38.2 Dependencies

FlareZonet requires dcsCommon and cfxZones to run

5.3.38.3 Module Configuration
Currently, FlareZones do not use config zones.

5.3.38.4 ME Attributes

Name	Description
flare	Tells DML that this trigger zone is a flare zone. The value of this attribute specifies the flare's color. Valid values are "green", "red", "white", "yellow", "random", "rnd", or "-1" (for random), "0" (for green), "1" (for red), "2" (for white) or "3" (for yellow) Defaults to "green" **MANDATORY**
rndLoc	If set to true, the flare is launched from a random location inside the flare zone. Defaults to false (flare is launched from center of the zone)
launch? launchFlare? f?	Watchflag. When the value of this flag changes, the flares are launched. If no flag is defined, no flares can be launched. Defaults to <none>
triggerMethod flareTriggerMethod	DML Method by which the flag is triggered. Defaults to "change"
direction	Compass heading in degrees (or a range) in which to launch the flare. The heading must be a positive number. When you supply a range, the direction will be randomized between the lower and upper bounds. Note that both numbers in the range must be positive numbers, and the upper bound must be larger than the lower. So if you want the flares to launch from 270 to 90 degrees, you need to add 260 to the second number. Examples: • 45 (launch in NE direction) • 90-180 (launch somewhere between East and South) • 270-450 (launch between West and East) Defaults to 0 (North)
altitude flareAlt agl	Height (in meters) above ground where the flare is launched from. Defaults to 1 (m above ground)
salvo	Number of flares to launch when the zone is triggered. Can be range. If you supply a range, a random number is chosen inside the range. Examples: • 3 (launch 3 flares • 2-5 (launch from 2 to 5 flares) Defaults to 1
duration	Only applicable when salvo contains more than 1 flares. The time (in seconds) over which the salvo is to be launched. If, for

Name	Description
	example salvo size is 4 flares, and duration is 2 seconds, a flare will be launched every 2/4 = 0.5 seconds, so that all 4 flares are launched of the span of 2 seconds. Duration can be a range, in which case the duration is randomized to a value inside the range. Defaults to 1 (second)

5.3.38.5 Using the module

To use flareZones in your mission, add the script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME where you want the flare(s) to be launched
- Add the ‘flare’ and trigger attribute to that zone.
- Add other attributes to customize to your needs

5.3.39 playerZone

5.3.39.1 Description

Player zone is a utility module that counts the number of players currently inside the zone, and that can generate events on the outputs when the number changes.



Additionally, it can set flags to the current number of players per faction, and the total number of players currently in the game.

5.3.39.2 Dependencies

playerZone requires dcsCommon, cfxZones.

5.3.39.3 Module Configuration

To change playerZone's global options,

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

Name	Description
red#	Name of the flag(s) to set to the current number of players that belong to the RED faction that are currently in the game. This value is updated once per second
blue#	Name of the flag(s) to set to the current number of players that belong to the BLUE faction that are currently in the game. This value is updated once per second
neutral#	Name of the flag(s) to set to the current number of players that belong to the NEUTRAL faction that are currently in the game. This value is updated once per second
total#	Name of the flag(s) to set to the current number of players (any faction) that are currently in the game. This value is updated once per second

5.3.39.4 ME Attributes

Name	Description
playerZone	Tells DML that this trigger zone is a player zone. The value of this attribute specifies which faction's players are counted. Possible values • "0" or "neutral" – red and blue are counted

Name	Description
	• "1" or "red" – only red players are counted • "2" or blue – only blue players are counted Defaults to "neutral" (red and blue players are counted) **MANDATORY**
method pwMethod	DML output method. Defaults to "inc"
pNum#	Direct output: the number of qualifying player units that are inside the zone at this moment. Defaults to <none>
added!	Output to bang! when players units have entered the zone since the last time the zone checked. Note that there are edge cases where added! and removed! can signal a change even though the flag pNum remains the same: for example if at the same time one player unit leaves the zone (or is killed), and another player unit enters the zone. Defaults to <none>
removed!	Output to bang! when plyer units have left the zone since the last time that the zone has checked. Note that there are edge cases where added! and removed! can signal a change even though the flag pNum remains the same: for example if at the same time one player unit leaves the zone (or is killed), and another player unit enters the zone. Defaults to <none>

5.3.39.5 Using the module

To use playerZone in your mission, add the script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME in which players are counted
- Add 'blue' or 'red' as value if you only want to count players from that side
- Add other attributes according to your needs

5.3.40 TACAN

5.3.40.1 Description

Similar to the NDM module, this module allows you to place TACAN beacons on the map, with all the bells and whistles you would expect from a DML module:





BASIC FUNCTIONALITY

In its most basic function, you simply place a trigger zone, and add the TACAN attributes (channel, callsign, mode), and you are set. A TACAN is placed at the center of the zone at mission start.



In-game a TACAN unit is placed on the ground, and any TACAN-equipped plane can use the TACAN signal to home in on it and perform approaches on any radial.

If the unit is destroyed, the TACAN signal ends, and the station can no longer be approached, so destroying enemy TACANs can be viable mission goal (see ‘tacan tracking’ for more information).

DEPLOY / SHUT DOWN (DESTROY) ON DEMAND

TACANS from DML can deploy and shut down (disappear) on demand, controlled by flags. When you tell a tacan zone to deploy their TACAN, it will usually first remove the last TACAN that it spawned to simulate a re-supply/refreshment or (when the location is randomized) relocation. Similarly, you can deactivate a deployed TACAN, which will destroy the deployed TACAN unit. Note that only the last deployed TACAN unit will be destroyed (see below when this can be an issue)

For advanced uses, removal before deployment can be disabled. In this case, new TACANs can be deployed without their former ‘incarnation’ being destroyed. Be advised that similar to cloners, when you tell a tacan zone to destroy the TACAN, only the last deployed TACAN is destroyed, so when you disable destroy before deply (by setting `preWipe` to false) it is possible that other active TACANs remain in the zone

RANDOMIZATION

The TACAN module supports heavy randomization: location, channel, callsign. For many mission applications within the confines of DML, randomization of a TACAN’s properties can

be counter-productive, as a TACAN often is only useful if you know which channel it is broadcasting on, and usually, they are placed in strategic locations (e.g., a runway, or to guide incoming CAP onto a target location). Therefore, most of the randomization options are primarily useful for mission designers who access the module via API. That being said, all randomization capabilities are accessible to all mission designers.

ANNOUNCED CHANGES / COALITION
Tacan can announce the placement and decommissioning of TACANs during the mission. This can be controlled individually per tacan zone.

LIST / DIRECTORY
The Tacan module can optionally provide a list per faction of known/available TACANs placed on the map via TACAN zones. TACANs that are destroyed or removed no longer appear on the list.

TACAN TRACKING
Since TACANS can be strategic targets, mission designers may want to track their destruction with the Group Tracker module. Each tacan zone provides an easy interface to pass a newly spawned TACAN unit to group trackers.

5.3.40.2 Dependencies
Tacan requires the modules dcsCommon and cfxZones.

Optionally: groupTracker

5.3.40.3 Module Configuration
To configure the tacan module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false
list GUI	Setting this attribute to ‘true’ adds a menu into communications → Other → Available TACAN stations to all players that automatically lists all TACANs deployed by neutral or their own faction. Only lists TACANS placed via tacan zones. When requesting a list, a player receives a list of all currently live TACANS placed by their or the neutral faction. Defaults to false (no tacan interface)

5.3.40.4 ME Attributes

Name	Description

Name	Description
tacan	Tells DML that this trigger zone is a tacan zone. The value of this attribute specifies which faction owns the TACAN. Possible values • “0” or “neutral” • “1” or “red” • “2” or blue Defaults to “neutral” (TACAN is owned by neutral faction, listed for red and blue factions when TACAN GUI is enabled) **MANDATORY**
channel	The channel to use for the TACAN, e.g. “72”. For randomization, supports lists and ranges, e.g. “12, 17, 42-47” Defaults to “1” (The channel 1)
mode	Either “X” or “Y” Mode “Y” usually is reserved for airborne TACAN (currently not supported with tacan zones) Defaults to “X”
callsign	Three-letter callsign for this TACAN station, e.g. “TCN”. For randomization, supports lists, e.g. “TXN”, “ABC”, “XYZ” Defaults to “TXN”
onStart	When set to true (default) the TACAN is created on mission start. If set to false, you must provide a “deploy?” attribute to be able to deploy the TACAN while the mission is running. Defaults to true (deploy on mission start)
heading	Orientation (in degrees) of the physical unit that is placed in game when the TACAN is deployed. Has no influence on the TACAN’s function. Defaults to 0 (oriented North)
rndLoc	When set to true, the TACAN is placed in a random location inside the trigger zone. Supports polygonal trigger zones. Otherwise, the TACAN is placed at the exact center of the trigger zone. Defaults to false (place at center of zone)
triggerMethod	DML method that tells when an input should trigger. Defaults to “change”
deploy?	Watchflag that triggers on triggerMethod and then initiates a deployment cycle. If preWipe is set to false, the previously deployed TACAN remains, otherwise it is removed before the new TACAN deploys. Defaults to <none>
preWipe	If set to true (default), the last deployed TACAN from this zone is removed (if it exists) before a new TACAN is deployed. Defaults to true (remove last spawn before deploy)
destroy?	Watchflag that triggers on triggerMethod and then removes the last previously deployed TACAN if it still exists. Defauts to <none>
c#	Output (direct). Is always set to the currently selected channel of the last spawned TACAN. When that TACAN is destroyed via destroy? Signal, the cannel is set to 0 (zero) Defaults to <none>
announcer	When set to true, any TACAN created with this tacan zone is announced with callsign, channel and mode to all players (when

Name	Description
	owning faction of the TACAN is neutral) or all players of the same faction that also owns the TACAN. Note that TACANS that are created at mission startup (onStart = true) are not announced. Defaults to false (no announcement)
trackWith:	When a TACAN is created, it is tracked with the groupTracker modules that are listed as value. Supports lists. Note that the groupTracker module must load before the tacan module in order to work correctly. Defaults to <none>
desc	(cosmetic only) – description to appear when this tacan is listed in the TACAN GUI. Requires that GUI is active Defaults to <none> (no additional description)

5.3.40.5 Using the module

To use the tacan module in your mission, add the script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME
- Add a 'tacan' attribute and optionally set blue or red as owner.
- Add other attributes according to your needs (recommended: channel and callsign

5.3.41 (Simple) Mission Restart

5.3.41.1 Description

Restarting a mission is a relatively simple affair in DCS, provided that you do not change the mission's file name later. If you do (as many mission developers tend to do, for example to include the version number), it becomes quite tedious. missionRestart is a tiny script that you can add to your mission that restarts the mission when you change the value of the flag named "simpleMissionRestart" to anything other than 0 (zero), no matter how often you changed the mission's file name.

That's it. So, if you want to restart the mission at any time, simply change the flag "simpleMissionRestart", and the mission restarts.

NOTE:
Current implementations of DCS require that any mission that uses this script runs as multiplayer, even when designed as a single-player version. It's expected that in the near future DCS runs all missions as multiplayer by default, as there are few downsides to running any mission as "local multiplayer", and performance is usually better, especially for AI-heavy missions. So, you might start yourself right now

5.3.41.2 Dependencies

(Simple) missionRestart has no dependencies.

5.3.41.3 Module Configuration

(Simple) missionRestart requires no module configuration.

5.3.41.4 ME Attributes

(none)

5.3.41.5 Using the module

Run the script during mission start-up as a DOSCRIPT action.

Then, when it is time to restart, change the flag 'simpleMissionRestart' to any other value than zero.

5.3.42 airfield

5.3.42.1 Description

The airfield module allows mission designers to query faction ownership of airfields/FARPs, generate signals when the airfields/FARPs are conquered, and they allow mission designers to arbitrarily force ownership of an airfield/FARP to a faction (for example, you can hand over ownership of Senaki-Kolkhi). This module is especially helpful when you need cloners on airfields to spawn clones that automatically belong to the airfield-owning faction, or when you want to force ownership of an airfield (or FARP) after some event without having to also place the required 'boots' on the ground first.

BASIC FUNCTIONALITY: Reporting Ownership/Change in Ownership

For some mission purposes you need a **signal/event when an airfield changes** hands (i.e. is conquered by another faction), or you may need the currently owning faction of an airfield as an input (for example for cloners). Airfield Zones attach themselves to ("associate with") the nearest airfield (or, optionally, airfields and FARPs), and they provide outputs that generate a signal when red or blue conquer it with the `red!` and `blue!` outputs. If you have an airfield and FARP in close proximity, make sure to verify that the airfield zone associates itself with the airfield/FARP that you intended (by enabling verbosity during testing).

An output `ownedBy#` always reports the currently owning faction of the associated airfield/FARP. You can use this direct output for a variety of purposes to easily implement otherwise difficult uses: for example, to trigger something each time that the airfield changes hands, or count how often an airfield has changed hands.

Also, **airfield zones always assume the same owner as the airfield/FARP that they associate with**. Modules that can align themselves with zone ownership (e.g. cloners, spawners) can use this to their advantage.

VISUAL CUES

Airfields can mark an airfield's capture zone (the 2km circular zone emanating from the airfield's center in the color of your choosing (usually defaults to red/bue/grey). You can optionally extend this ability to all airfields on your map, or only those that have an 'airfield' zone attached.

SETTING AIRFIELD OWNERSHIP

Airfield zones can force (set) ownership of the associated airfield/FARP at any time. For this you can use the `makeRed?` and `makeBlue?` inputs, or use the 'fixed' attribute. The "fixed" attribute forces ownership of the associated airfield/FARP at mission start to the coalition you list, and then disables capture until a signal is received on the `autoCap?` input, which revokes its 'unconquerable' status.

If the zone receives a signal on the `makeRed?` or `makeBlue?` inputs, it first wrests ownership control from the mission (meaning that the airfield/FARP can no longer be captured by 'ground rules') and then gives ownership to the respective faction. Ownership remains with that faction until

- a different `makeXXX?` signal is received
- an `autoCap?` signal is received

If the zone receives an `autoCap?` signal, the associated airfield reverts to standard mission capture “ground rules”.

If you want to inhibit an airfield’s/FARP’s capture from the start of the mission until you decide otherwise, you can add a `directControl = true` attribute to the airfield zone. Such a zone can’t be captured normally until it receives an `autoCap?` signal

If an airfield is captured by sending a signal on a `makeXXX?` input, a capture signal is sent on the corresponding `xxx!` Output, but only if the airfield belonged to a different faction before.

PERSISTENCE SUPPORT
airfield zones fully support persistence.

5.3.42.2 Dependencies
Airfield requires the modules dcsCommon and cfxZones.

If you are using persistence and cloners or spawners that take their ownership from airfield zones, airfield should load before cloneZones and spawn zones.

5.3.42.3 Module Configuration
To configure the Airfield module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Defaults to false
redFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the airfield’s capture zone when owned by RED coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to #FF000033, a deep red, 80% transparent
redLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the airfield capture zone’s outline when owned by RED coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote

	colors) Defaults to #FF0000FF, a deep red, 100% opaque
blueFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the airfield's capture zone when owned by BLUE coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to #0000FF33, a deep blue, 80% transparent
blueLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the airfield's capture zone when owned by BLUE coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to #0000FFFF, deep blue, 100% opaque
neutralLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the airfield capture zone's outline when owned by NEUTRAL coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to #CCCCCCFF, light grey, 100% opaque,
neutralFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the airfield's capture zone when owned by NEUTRAL coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to #CCCCCC33, light grey, 80% transparent
show	When set to true, all airfields (in addition to those flagged with 'how' in their associated 'airfield' zones) are shown with their capture zone colored in their current faction. Defaults to false (show only those capture zones that have an associated 'airfield' zone with their 'show' attribute set to true).

5.3.42.4 ME Attributes

Name	Description
airfield	Tells DML that this trigger zone should associate with the closest airfield (or FARP, see config settings) **The owner of this zone is always the same as the owner of the airfield/FARP that an airfield zone associates with.**

Name	Description
	The value of this attribute is ignored. **MANDATORY**
farps	When set to true, airfield zones also associate with FARPs. When set to false, FARPs are ignored. When enabled, care must be taken when placing FARPS close to airfields: an airfield zone associates itself with the *closest airfield or FARP*, and the in-game location of an airfield may not be where you assume it is. Hint: enable verbosity and see which airfield zone attaches to which object. Defaults to false (associate only with airfields, disregard FARPs)
fixed	Defines which side holds the associated airfield/FARP **at mission start.** The airfield/FARP is made **unconquerable**, and remains in that coalition's possession until the zone receives a signal on one of the following inputs: • `makeRed?` – ownership is turned over to RED • `makeBLUE?` – ownership is turned over to BLUE • `makeNeutral?` – ownership is turned over to NEUTRAL • `autoCap?` – makes the airfield/FARP capturable by mission ground capture rules. Valid values are 0, 1, 2, red, blue, neutral Defaults to <none>, airfield can be captured normally, and initial ownership is as set up by mission editor.
method	Output method for all outputs. Defaults to 'inc'
triggerMethod	DML input method for all inputs Defaults to 'change'
red!	Flags that should be sent a signal when the associated airfield/FARP is captured by RED faction Defaults to <none>
blue!	Flags that should be sent a signal when the associated airfield/FARP is captured by BLUE faction Defaults to <none>
makeRed?	When a signal is received on the flag that connects to this input, the zone takes control of airfield/FARP ownership (no longer mission ground capture rules), and forces ownership to RED faction. If the airfield/FARP was previously owned by a different faction, a signal is sent to red! Output. To have the airfield return to being able to be captured by other factions, send a signal to the autoCap? input. Defaults to <none>
makeBlue?	When a signal is received on the flag that connects to this input, the zone takes control of airfield/FARP ownership (no longer mission ground capture rules), and forces ownership to BLUE faction. If the airfield/FARP was previously owned by a different faction, a signal is sent to blue! Output. To have the airfield return to being able to be captured by other factions, send a signal to the autoCap? input. Defaults to <none>

Name	Description
makeNeutral?	When a signal is received on the flag that connects to this input, the zone takes control of airfield/FARP ownership (no longer mission ground capture rules), and forces ownership to NEUTRAL faction. To have the airfield return to being able to be captured by other factions, send a signal to the autoCap? input. Defaults to <none>
autoCap?	When a signal is received on the that connects to this input, the zone relinquishes ownership control over the associated airfield/FARP and ownership is determined by the mission ground capture rules Defaults to <none>
directControl	When set to true, at mission start the zone assumes control over the associated airfield/FARP's ownership: it can no longer be captured by mission ground capture rules. Use the `autoCap?` input to relinquish control back to the mission, or `makeRed?`, `makeNetral?` and `makeBlue?` inputs to explicitly set ownership. Defaults to false (at mission start, the associated airfield's ownership is determined by mission ground capture rules)
ownedBy#	Flags connected to this output are set to the value of the currently owning faction of the associated airfield/FARP: 0 (neutral), 1 (red) or blue (2) Defaults to <none>
redFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the airfield's capture zone when owned by RED coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to config zone's "redFill" value
redLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the airfield capture zone's outline when owned by RED coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to config zone's 'redLine' value
blueFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the airfield's capture zone when owned by BLUE coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to config zone's "blueFill" value
blueLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the airfield capture zone's outline when owned

Name	Description
	by BLUE coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to config zone's 'blueLine' value
neutralFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the airfield's capture zone when owned by NEUTRAL coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to config zone's "neutralFill" value
neutralLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the airfield capture zone's outline when owned by NEUTRAL coalition. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to config zone's 'neutralLine' value
show	If set to true, the airfield's 2km capture zone is shown on the F10 map in the colors described above, by owning coalition. Defaults to false (capture zone is not shown)

5.3.42.5 Using the module

To use the airfield module in your mission, add the script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME close to the airfield (or FARP) that you want to associate the zone with (to control FARPS, the config zone must be set up accordingly)
- Add an 'airfield' attribute
- Add other attributes according to your requirements

5.3.43 ownAll

5.3.43.1 Description

ownAll is a coordination/helper module that pulls together, and consolidates, information from multiple modules that deal with zone ownership - which is usually a major ingredient in deciding win conditions. In short, it provides 'ownership overview' across multiple zones that include

- Owned zones
- FARP Zones
- Airfield Zones
- ownAll (for cascades)

Use ownAll zones whenever you want to be informed when a faction owns a specific set of zones.

BASIC FUNCTIONALITY

To set up, you give an ownAll zone a list of zones that it monitors. Given this set of zones, an ownAll zones continuously gives you a tally of how many of these zones are owned by red, and how many are owned by blue.

Should one faction manage to own all zones in the list, ownAll can give a signal on the output for that particular faction.

Importantly, should one faction assume ownership of all zones that it monitors, **it sets it's own owner to that faction**, allowing you to cascade ownAll zones into ownership hierarchies.

Compared to OwnedZones or other module that manage zone ownership, ownAll has a number of important abilities:

- *Set-based*
  each ownAll zone operates on a set of zones that you, the mission designer, defines OwnedZones always regards *all* owned zones over the entire map.

- *Cascading*
  ownAll zones will take on the ownership of whoever owns all zones listed in their set. This you can use to set up cascading groups of zones that can/must be captured by referencing other ownAll zones in the list of zones to monitor.

- *Module-agnostic*
  Since ownAll uses sets/lists of zones, you can use ownAll to pull together other zones with managed ownership independent of their type: they can be ownedZones, FARP Zones, Airfield Zones

5.3.43.2 Dependencies

ownAll requires the modules dcsCommon and cfxZones.

5.3.43.3 Module Configuration
To configure the ownAll module via a configuration zone,

- Place a Trigger Zone on the map in ME
- Name it “ownAllConfig” (note: name must match exactly)
- Add any of the following attributes to this zone (currently none defined)

Name	**Description**
verbose	Show debugging information. Defaults to false
redLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone’s outline when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Defaults to “1.0, 0, 0, 1.0”
redFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Defaults to “1.0, 0, 0, 0.2”
blueLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone’s outline when it belongs to the BLUE faction. RGBA values each range from 0.0 to 1.0 Defaults to “0, 0, 1.0, 1.0”
blueFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the BLUE faction. RGBA values each range from 0.0 to 1.0 Defaults to “0, 0, 1.0, 0.2”
neutralLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone’s outline when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Defaults to “0.8, 0.8, 0.8, 1.0”
neutralFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Defaults to “0.8, 0.8, 0.8, 0.2”

5.3.43.4 ME Attributes

Name	Description
ownAll	List of the names of zones, separated by comma "," that must be owned by the same faction and are monitored throughout the mission. Example: `Zone A, Another Zone, abc` You should include at least two (2) zones to the list, else you will receive a warning. Supports all trigger zones, including (especially, really) DML zones with managed ownership like Airfield Zones, Owned Zones and FARP Zones. Since an ownAll zone manages its own ownership, you can also include ownAll zones to this value. Do this to create a cascade (or hierarchy) of ownership. **Mandatory**
red#	A direct output that represents the number of zones from the list that are currently held by RED faction Defaults to <none>
blue#	A direct output that represents the number of zones from the list that are currently held by BLUE faction Defaults to <none>
total#	A direct output (set only at the beginning of the mission) that represents the number of zones that are monitored
red!	Output to send a signal on when RED possess **ALL** zones that are listed Defaults to <none>
blue!	Output to send a signal on when BLUE possess **ALL** zones that are listed Defaults to <none>
method	DML method for outputs (excluding direct outputs)
show	When set to true, the associated airfield/FARP is shown in the map with a 2 km wide circle on the map denoting the 'capture perimeter'. The circle is filled with the owning faction's color (see below) Defaults to false, capture perimeter not shown.
redLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone's outline when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Defaults to "1.0, 0, 0, 1.0"

redFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the RED faction. RGBA values each range from 0.0 to 1.0 Defaults to “1.0, 0, 0, 0.2”
blueLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone’s outline when it belongs to the BLUE faction. RGBA values each range from 0.0 to 1.0 Defaults to “0, 0, 1.0, 1.0”
blueFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the BLUE faction. RGBA values each range from 0.0 to 1.0 Defaults to “0, 0, 1.0, 0.2”
neutralLine	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone’s outline when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Defaults to “0.8, 0.8, 0.8, 1.0”
neutralFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone when it belongs to the NEUTRAL faction. RGBA values each range from 0.0 to 1.0 Defaults to “0.8, 0.8, 0.8, 0.2”

5.3.43.5 Using the module

To use the airfield module in your mission, add the module’s script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME
- Add an ‘ownAll’ attribute and list the names of the trigger zones that should be monitored as value, separated by comma “(“,”)
- Add other attributes as required.

5.3.44 groundExplosion

5.3.44.1 Description
GroundExplosion is basically DML's take on ME's 'EXPLODE' trigger rule action, except that it greatly expands on the concept, and fully integrates with DML's concept of communication with flags.

BASIC FUNCTION
With groundExplosions you can use other modules to trigger explosions on the ground or near the ground ('Flak'). When the module is triggered via the "boom?" input, it creates

- a single or series of explosions
- of a controllable strength
- at the center ort randomized location inside the trigger zone
- at controllable altitude
- (if multiple explosions are selected) over a customizable duration

Using these attributes, you can use the module to

- Destroy map/scenery objects at will
- Create (dangerous!) "Flak" – a series of explosions inside a volume over time
- Simulate bombardments on the ground
- Tightly controlled special effects for mission 'flavor'



NUMBER OF EXPLOSIONS
The "num" attribute controls how many explosions the module will create each time that it is triggered. If, for example, you supply "3" as value, the module will create three explosions every time that 'boom?' is triggered. "nun" supports randomized ranges, meaning that if you supply the range "3-7" for "num", each time that "boom?" is triggered, the module picks a number between 3 and 7 (inclusive) and then creates that many explosions.

STRENGTH
You control the strength of the explosion through a value added to the “explosion” attribute itself. 1 is a fairly weak explosion, while 3000 usually is sufficient to level most buildings (and their neighbors). groundExplosion supports randomization of explosion strength: if you supply a range (e.g., “500-750”), the strength will be randomized to a value from that range for every explosion created.

RANDOMIZED LOCATION
When “num” (the number of explosions to create) is set to any value larger than 1, or if you set the “rndLoc” attribute to true, the location(s) of explosion are randomized to points inside the trigger zone. Circular and Quad-based trigger zones are supported.

ALTITUDE
Although named “groundExplosion”, you can control the altitude at which the explosions occur. The number that you specify here is always AGL in meters, and the attribute supports ranges. In combination with the ‘num’ attribute, you can use this to quickly and easily create a volume filled with explosions that is visually indistinguishable from WWII ‘Flak’ – and which is also very dangerous to any aircraft within the volume.

DURATION (multiple explosions only)
Whenever you want more than one explosion to occur inside the trigger zone, you can also set a time period over which these will occur, for example you can set up 6 explosions to occur over the span of 2 seconds.

FLARES
Some explosions (e.g. munitions that are cooking off) have also flares erupting. If you want this additional eye candy (it’s only a visual effect), you can add the ‘flares’ attribute. As a value you give the number of flares that each explosion should set off. flares supports a range, which will randomize the number of flares given off by each explosion to a value between those two numbers.

5.3.44.2 Dependencies
groundExplosion requires the modules *dcsCommon* and *cfxZones*.

5.3.44.3 Module Configuration
groundExplosion currently has no global settings and does not require a config zone.

5.3.44.4 ME Attributes

Name	Description
explosion	Marks this zone as an explosion zone. All explosions triggered will occur within the zone (although their damage/blast may well extend

Name	Description
	past the zone. The value of this attribute determines the strength of the explosion. A value of 1 may be harmful to individual units, while a value of 3000 is almost sure to level a building (and others close by, depending on their strength) Supports ranges, so if you specify a strength of 50-230, each individual explosion will receive its own randomized strength within that range Defaults to 1 **MANDATORY**
triggerMethod	DML trigger method for inputs. Defaults to “change”
boom?	DML watchflag. When triggered, the module runs through an explosion cycle, setting of as many explosions as specified by the attributes Defaults to <none>
num	Number of explosions to perform every time that the input is triggered. Supports ranges, so setting *num* to “3-9” will cause groundExplosion to set off a random number between 3 and 9 (inclusive) each time that the input is triggered. Defaults to 1 (a single explosion)
rndLoc	Controls where the explosion will be located. If set to false (default) the explosion will happen at the exact center of the trigger zone. If *rndLoc* is set to true (or *num* is greater than 1), the positions of all explosions are always randomized to occur within the area of the trigger zone. **NOTE:** If *num* is set to a value greater than one (1), the location of each explosion is always randomized inside the zone. Defaults to false (if a single explosion is chosen, it occurs at center of zone)
agl	Height (in meters) above local ground where the explosion occurs. This allows you to create air bursts, and even Flak effects. **Important:** if your target zone spans large areas with steep inclines, this can result in explosions with great variance in height, as the explosion always occurs *agl* meters above the local ground. Can be set to a random range (e.g. 20-30) that is applied to each explosion individually

Name	Description
	Defaults to 1 (m above ground)
duration	Only applicable if set to multiple explosions. Sets the time frame (in seconds) for all explosions to occur within, after being triggered. So if you set *num* to 8 and *duration* to 2, all 8 explosions occur within a span of 2 seconds. Defaults to 0 (all explosions happen immediately at time of triggering)
flares	Adds flares that erupt from the explosion. The value that you supply controls the number of flares that are launched with each explosion, and supports ranges (e.g. “1-3”). Note that flares are always launched from the ground, so this effect may look silly when used with *agl*. Defaults to <none> - no flares

5.3.44.5 Using the module

To use this module in your mission, add its script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME
- Add an ‘*explosion’* and a ‘*boom?*’ attribute, then
- add other attributes as required.
- The explosions are triggered with the flag that is connected to the *blow?* input

5.3.45 Sweeper

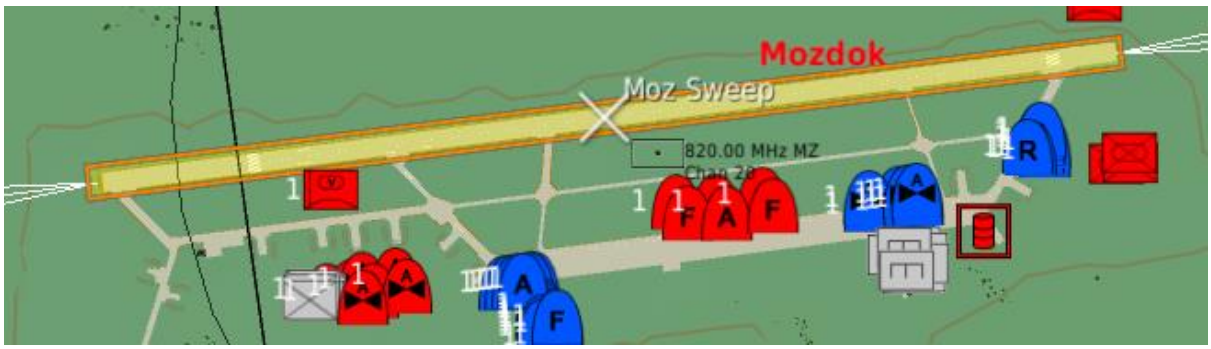
5.3.45.1 Description

In DCS, stuff blows up. And sometimes, some of that stuff can obstruct runways. In most missions, this isn't much of a problem, but in rare instances it can lead to issues when they obstruct key/choke points that AI units have to navigate. Missions that are long-running are more susceptible than short. If wayward units, debris, burnt-out husks, even AI aircraft obstruct progress of your mission, use Sweeper to ensure that your choke points are cleared out regularly. There usually are only a few locations in your mission that must be swept, and sweeping is a process that can be done with large (multi-minute) gaps in-between.

So, when you test your mission design, and you notice that units (e.g. AI flights) are queuing up instead of being on their way, chances are good that something obstructs their path. For example, a SAM site erected to protect an airfield may be attacked by the enemy, and the SAM site units disperse. Since AI in DCS isn't, one or more of these units may come to rest on a runway, blissfully unaware of the fact that it now prevents all AI departures and landings at that airfield. Sweeper can remedy this for you.

How Sweeper Works

Using sweeper is simple: set up a sweeper trigger zone, and any unit that remains inside that zone longer than the sweeper interval is removed from the game. That's all. Simply being inside a sweeper zone isn't an issue for sweeper. Being inside the zone longer than the sweep interval will cause sweeper to remove it.



Above image shows how a trigger zones "Moz Sweep" placed over Mozdok's runway protects the runway from becoming inoperable should objects fall on the runway. Example taken from the "Expansion" mission.

5.3.45.2 Dependencies

Sweeper requires *dcsCommon* and *cfxZones*.

5.3.45.3 Module Configuration

To configure the sweeper module via a configuration zone,

- Place a Trigger Zone on the map in ME
- Name it "sweeperConfig" (note: name must match exactly)
- Add any of the following attributes to this zone (currently none defined)

Name	Description
verbose	Show debugging information. Defaults to false
interval	Time (in second) between to sweeper checks of a zone. Any unit that is found inside the zone in two consecutive checks is immediately removed from the game. Note that these checks can be costly performance-wise, so multi-minute intervals are to be preferred. Note also that if your mission has units traverse these zones, make sure that the interval is large enough that the traversing units have time to clear the zone (for example, some taxiway-less airfields can force aircraft to backtaxi for take-off – a process that can easily take multiple minutes) Defaults to 300 Seconds (5 Minutes)

5.3.45.4 ME Attributes

Name	Description
sweeper	Marks this zone as a sweeper zone. No unit may remain longer than interval (see above) inside the zone or it risks being removed from the game **MANDATORY**
aircraft	If set to true, aircraft are included in the sweep and removed after being found inside the zone on two consecutive checks Defaults to true (aircraft are included in sweeps)
helos	If set to true, helicopters are included in the sweep and removed after being found inside the zone on two consecutive checks Defaults to true (helicopters are included in sweeps)

5.3.45.5 Using the module

To use this module in your mission, add its script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME and place it over a choke point that must regularly be cleared of obstructions
- Add a '*sweeper'* attribute
- add other attributes as required.

5.3.46 twn (automatic Town locale bridge)

5.3.46.1 Description
Each map in DCS comes with it's own internal data base of location information (mainly town names and location) that was originally intended to be accessed by the Ka-50 (Black Shark) Module. No other DCS module currently accesses this information, and sadly it is not accessible through Mission Editor. But it comes with every map.

Using some black scripting magic and advanced code heresy, DML is now able to tap into this information source and allow other DML modules to access it.

5.3.46.2 Dependencies
None. This module uses magic.

5.3.46.3 Modules that automagically use twn when present
Some modules automatically look for, and if present, automatically integrate with twn's locale information, e.g.

- csarManager
- convoy
- recon mode
- messenger's "twn" and "loc" wildcard
- valet's "twn" and "loc" wildcards
- reaper

5.3.46.4 Module Configuration
None

5.3.46.5 ME Attributes
None

5.3.46.6 Using the module
Add the twn module to your miz.

5.3.47 Convoy

5.3.47.1 Description

This module allows you to easily create (helicopter-escorted) convoys that you can send across the map and receive signals when they arrive or are destroyed. The module also supports reports for active (friendly and enemy) convoys.



BUILDING A CONVOY: Ground and Helicopters

In this regard convoys function very similar to cloners: you build ground troops (of your own composition), place at least one unit inside the convoy zone, and define the route that they should take. Also set their route orders (ROE, Alarm state etc.). You can place multiple groups inside a convoy zone (with different compositions and routes) – and they are all added to the group of potential convoys that are to be sent out.

Next, if you want helicopters to escort the convoy, add one or more helicopter groups into the convoy zone and set up their armament and fuel.

When you activate the convoy, one of the ground groups and one of the helicopter groups are randomly chosen. The helicopters are always set to escort the ground group and will patrol around the moving convoy until they reach their goal.

DESTINATION AND COMPOSITION

A convoy is simply a group of ground units, and you assemble a convoy in Mission editor the same that you would any other ground group: choose the number and type of vehicles, then assign a path to the group and define any options (ROE, alert stage etc.) that governs the ir behavior.

Note that since you can have multiple groups inside a convoy zone, and groups can have different destinations, a convoy spawned from one convoy source can have different locations, depending on which group is chosen to spawn (a chosen group’s route/destination is never changed). So, if you have three convoys sitting in a convoy zone, named “A” and set up to go to WPA, “B” bound for WPB, and “C” driving to WPC), each time that a new convoy is spawned, the module chooses one group: “A”, “B” or “C”. By choosing a group, the

destination is set: if “C” is chosen, that convoy always heads to its WPC. In other words, routes are not exchanged between convoys.

Be advised that it doesn’t matter which coalition you choose inside ME to assemble a convoy group. The coalition that they belong to is entirely determined by the coalition of the convoy zone (see *coalition* attribute). For simplicity, I recommend that you use one of the combined joint task forces as source, so you gain access to all vehicles (changing countries while assembling a group is very, very buggy in Mission Editor).

ONLY GROUND UNITS RELEVANT

Although from a player’s perspective a convoy can be a mix of ground vehicles and helicopter, for DML only the ground units are relevant. If a ground vehicle is destroyed, a signal can be sent on the *attacked!* output. If an escorting helicopter is destroyed, no signal is sent. Likewise, a convoy is considered dead when all ground vehicles are destroyed, irrespective of how many helicopters remain.

STARTING A CONVOY

A convoy is started when the zone receives a signal on the *spawn?* input. When a convoy zone starts a new convoy, it

- randomly chooses one of the ground vehicle groups and spawns the group into the game. If *preWipe* is true, any previous incarnations of this convoy, including their helicopter escorts, are de-spawned.
- When a ground vehicle group is chosen, its route is analyzed to determine the destination. If the *twn* module is present, the nearest location’s name is retrieved, otherwise the name of the nearest trigger zone or airfield (whichever is closest to the last waypoint of the ground group’s route)
- if there are Helicopter groups defined, determines if it should add a helicopter escort by rolling the dice and comparing the result with *pEscort*
- If a helicopter escort is to be added, one of the helicopter groups is chosen by random, and their task is to aggressively defend the ground vehicles. The helicopters start in the air.

REPORTS AND GUI

The convoy module can provide its own menu to players that allows them to request status information of convoys. Additionally, it can provide real-time updates when a convoy reaches a waypoint, is under attack, or is destroyed. Mission designers can control what information is provided in-mission with convoy’s config zone, and who receives this information.

Real-Time messages

When enabled, players are told when a convoy

- spawns:
  Convoy name (e.g. “Blue Convoy-76544”) where it originates from (if twn is present with information about the locale), where it is headed (if twn is present, again with locale information) and the distance is is going to travel, e.g.



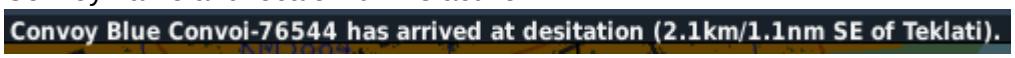
When enabled, the enemy coalition receives an abridged version of this notification.

- reaches a waypoint (unless it is the first or last Waypoint):
  Convoy name with destination (with locale information if twn is loaded), and which waypoint was reached out of the total number of waypoints. Waypoint 1 is the spawn point, the last waypoint is the destination, e.g.



  This information is never shown to enemy coalition

- reaches its destination
  Convoy name and locale if twn is active.



- is attacked (one or more units lost), if enabled
  Convoy name, destination (with locale if twn is loaded), location of attack (if twn is present)



  (Note: only one such attack warning is sent every 5 minutes, so if that convoy sustains more losses during the intervening time, no additional notices are sent.

- is destroyed:
  Convoy name and destination.



Note that players can also receive information about enemy convoys if that is enabled. However, in this the information is anonymized as DML assumes that that information isn't available:

- On spawn:
  An anonymized convoy name (e,g, "CVY-3"), no origin and a generalized destination (the destination is derived from the name of the nearest trigger zone name or airfield/FARP, whichever is closest).



- On Arrival
  Anonymized convoy name and generalized destination.



Reports
When enabled, players have UI access to intelligence/field reports on the status of known convoys. These reports can be requested from the Communication→Other… menu. When requested, the report lists and groups convoys be faction, with the mission designer being able to control if a faction can see information on enemy convoys through the config zone.



Enemy convoys are anonymized, with two important QoL additions to better implement 'convoy hunter' style missions:

If a side manages to destroy one or more ground units from an enemy convoy, that convoy will list the number of remaining units as it is assumed that the attacking unit(s) reported in their sighting.

If a convoy has reached the last leg of its route, that convoy's information adds the line "CLOSE TO DESTINATION" to alert players to the fact that this convoy may warrant higher prioritization.

DESTINATIONS

Since convoys derive their destination name for some notifications and reports from the name of the airfield/trigger zone nearest to the convoy's final waypoint, you may want to take advantage of this and place an (otherwise empty) trigger zone close to the last waypoint and give that trigger zone a fitting name (e.g. "Poti Defense Bunkers")

If the *twn* module is active, the destination information can be significantly amended by locale information.

ARRIVING AT DESTINATION

When a convoy reaches its last waypoint, it is considered as reaching its destination. In this case,

- a notification (text) is sent out
- the flag that connects to the output *arrived!* is triggered
- if the convoy zone's *endWipe* attribute is set, the convoy and escort are removed from the game.

OUTPUTS (SIGNALS)

- **attacked!**
  The convoy module periodically (every 10 seconds) checks all convoys. If it detects that a convoy has lost one or more ground units since the last check, and still has one or more units alive, it sends a signal on all flags that connect to the attacked! output. Note that the status of any escorting helicopter is ignored.

- **dead!**
  If the convoy module detect that all ground units (helicopters are ignored) of a convoy are destroyed, it sends a signal to all flags that connect to the *dead!* output.

- **arrived!**
  When DCS deems that the ground vehicle group for the convoy has arrived at their

final waypoint, the convoy module sends a signal on the *arrived!* output to all flags that connect to it.

5.3.47.2 Dependencies
Convoy requires *dcsCommon*, *cfxZones* and *cfxMX*.

Optional: *radioMenu* (when using the *"attachTo:"* attribute)

Convoy automatically integrates with *twn* if it is present in the mission

5.3.47.3 Module Configuration
To configure the Convoy module via a configuration zone,

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

Name	**Description**
verbose	Show debugging information. Defaults to false
actionSound	Name of the soundfile to play when the convoy module hast something to announce. Defaults to nothing
menuName	Name of the menu that the Convoy module installs for players if the module's GUI option is enabled. Defaults to "Convoys"
hasGUI	When set to false, Convoy's interactive menu is disabled Defaults to true (interactive menus are enabled)
listEnemy	When reporting convoys, include enemy convoys. Defaults to true (enemy convoys are included)
listNeutral	When reporting convoys, include neutral convoys. Defaults to true (neutral convoys are included)
attachTo:	Name of the radioMainMenu trigger zone to attach the convoy menu to. Only applicable if hasGUI is enabled. Requires that the radioMenu module has loaded. Defaults to <none>
redConvoy#	(Direct) output that is set to the number of red convoys that are currently alive. Defaults to <none>
blueConvoy#	(Direct) output that is set to the number of blue convoys that are currently alive Defaults to <none>
neutralConvoy#	(Direct) output that is set to the number of neutral convoys that are currently alive Defaults to <none>
allConvoy#	(Direct) output that is set to the total number of convoys (red + blue + neutral) that are currently alive Defaults to <none>

5.3.47.4 ME Attributes

Name	Description
convoy	Marks this zone as a convoy Zone. All Ground groups that have at least one vehicle inside this zone become convoys. You define the route and the orders for those groups using Mission Editor. If more than one ground vehicle group is inside the convoy zone, one is chosen by random each time the zone is signaled to spawn. All helicopter groups inside this zone become possible helicopter escorts. Only one group (or less, see *pEscort*) is chosen to escort the convoy in its way. Remember to set up the helicopter's loadout. **MANDATORY**
coalition	The coalition that the convoys spawned by this zone belong to. Defaults to NEUTRAL
dynamic	EXPERIMENTAL – do not use
Identical unique	EXPERIMENTAL – do not use
preWipe	When set to true, all previously spawned convoys are removed from the mission. Defaults to false (spawns do not remove previous convoy)
endWipe	When set to true, a convoy that reaches their last waypoint is removed from the map. This will prevent vehicles massing around the last waypoint when multiple instances of the same convoy reach their destination. Defaults to true (convoys are removed when they reach the last waypoint)
killWipeDelay	Delay (in seconds) for escorting helicopters to be removed after all ground vehicles of the convoy are destroyed. This allows you to have the escort helicopter loiter like 'angry hornets' for a while before they, too, are removed. Defaults to 300 (5 Minutes)
pEscort	Probability (in Percent) that the convoy is escorted by helicopters. This requires that at least one helicopter group is present inside the convoy zone. For example, if pEscort to 25, there is a 25% chance that a convoy when spawned also receives an escort. If pEscort is smaller than 1, that convoy never receives a helicopter escort. Defaults to 100 (always escort if helicopters are present)
onStart	If set to true, a convoy is spawned from this zone at the start of the mission. Defaults to false (no spawn on start)
wpUpdates	If set to true, coalitions receive an update each time that a convoy reaches a waypoint. There are separate messages for the first waypoint (when the convoy is spawned), last waypoint (at least one ground unit of the convoy reaches destination), and in-between waypoints. When a convoy reaches the penultimate waypoint (and assuming that there are at least 3 waypoints in the convoy's route), the enemy coalition also receives a separate notification. Defaults to true (waypoint updates are sent)
attackWarnings	If set to true, the module sends out warnings to the coalition that owns the convoy when it gets attacked and at least one unit is destroyed. Only one warning per attacked convoy is sent within 5

Name	Description
	minutes. Defaults to true (coalition receives warnings)
spawn?	Input. When triggered, this zone spawns a new convoy Defaults to <none>
dead!	Output: flag to change when all ground vehicles of a convoy spawned by this zone are destroyed. Defaults to <none>
attacked!	Output: flag to change when ground vehicles of a convoy have been destroyed since the last check, and the convoy isn't completely destroyed. Defaults to <none>
arrived!	Output: flag to change when the convoy (i.e. at least one living ground unit) has reached its destination. Defaults to <none>
convoyMethod	DML method that is sent to the output. Defaults to "inc"
convoyTriggerMethod	DML method that triggers the input. Defaults to "change"

5.3.47.5 Using the module

To use this module in your mission, add its script to the mission as a DOSCRIPT action during Mission Start.

To use

- Add a trigger zone in ME
- Add a '*convoy'* attribute
- add other attributes as required.
- Add at least one group of vehicles into the trigger zone, and define its route
- Optional: add helicopter groups that are to function as escorts and set up their loadout and fuel.

5.3.48 WHpersistence

5.3.48.1 Description
This module requires the “persistence” module.

Similar to the unitPersistence module, this module transparently saves and restores the current status (inventory) of all warehouses in a mission. If you don’t work with warehouse inventories, you also don’t need this module.

This module currently has no interface for mission creators, it has an all-or-nothing approach: add it to the mission, and the mission persists or loads the inventory of all warehouses every time that persistence is invoked. Don’t add it to a mission, and inventories are ignored by persistence.

5.3.48.2 Dependencies
Requires dcsCommon, cfxZones, and persistence.

5.3.48.3 Persistence
This module persist/loads all inventories of all warehouses.

5.3.48.4 Module Configuration
None.

5.3.48.5 ME Attributes
None.

5.3.48.6 Using the module
Add WHpersistence to your mission via a DOSCRIPT action at MISSION START

5.4 “Big Ticket” (“drop-in”) Modules

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

5.4.1 Player Score / PlayerScoreUI

5.4.1.1 Description

Player Score is a drop-in module for score keeping with a host of features:

- Score is kept separately for players and factions. Players can change their faction during missions without upsetting their (or the faction’s) score
- Tabulating kills and points awarded for kills
- Restrict scoring to zones, so kills outside designated zones don’t count
- Keeping mission-specific “feats”
- Awarding of kills and feats can be deferred until players land in designated ‘score safe’ zones.
- Mission designers can design feats; feats support extensive wildcards (like messenger) for individualized feats
- Feats can be limited in how many times they can be achieved
- Mission designers can supply triggers that add score to a faction on-demand
- Supports persistence
- Score table can be exported to a plain text file
- Automatically integrates with the “bank” module and converts score points into funds for the pilot’s coalition.

In a nutshell, PlayerScore automatically keeps score and a “Kill/Feats log” for each player and coalition. 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 player 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. PlayerScore 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/Feat log.

Awarding scores can be deferred until the player successfully land their aircraft in specially designated ‘safeScore’ zones. Should they change aircraft or crash, their accumulated scores since last successfully landing are discarded.

PlayerScore also allows the current score (entire table) to be saved as plain text-

Independent Player and Coalition tracking, “griefing” prevention

PlayerScore keeps track of the player’s score individually, and separate from the coalition. This means that players can switch coalitions and increase their score in the same game

without upsetting a coalition score (the individual score does carry over for the player, but is not added to the coalition). Likewise, to prevent griefing, a negative score only affects an individual's score, and is not added to the coalition. This prevents players from switching to the other side and commit atrocities with the intent of lessening the other faction's score.

If you are certain that griefing cannot happen in your missions, you can use the the *noGrief* = *false* attribute in the config zone to disable grief prevention. In that case, negative scores also affect the coalition score.

Kill Zones can restrict scoring

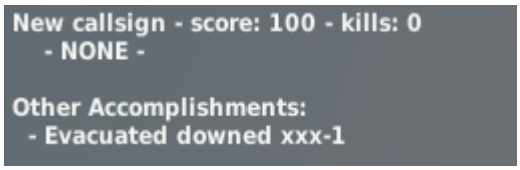
By default, playerScore scores all kills as the happen on the map. However, mission designers may want to restrict scoring kills only to certain aeras on the map – for example to specially designated combat zones, or to prevent players 'camping' enemy spawn locations for easy kills. To restrict scoring of kills only to certain zones, all you need to do is add one or more "Kill Zones" to the mission. Once playerScore detects the presence of at least one kill zones in your mission, all scoring of kills is restricted to these zones. In order to score a kill, either the killed unit or both units (player and target, when 'duet' attribute is set for the kill zone) must be inside the kill zone at the point of kill.

Note that **kill zones only affect scores and feats connected to kills**. Other feats can be accomplished outside of kill zones.

Announcing Kills

Player Score announces each kill with the score and current total for each side. This feature can be turned off with the *announcer* attribute in the config zone. Player to AI (PvE) 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 Kill 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.

WARNING:
As of DCS version 2.7, the game engine internally can change a unit that is 'cooking off' from ground unit to static object. This happens unpredictably, and when it happens, the unit in question loses access to the group that it belonged to. This quirk in DCS can lead to problems with correct scoring when your score table assigns score by group name. In cases like this, PlayerScore usually falls back to its default score table (see below)

If the unit killed isn't mentioned on the *playerScoreTable* (neither unit name, group name, nor type), a default score is used (see below).

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 any points. Phrased differently: to award a score for a static object or scenery object, that object must be included in the *playerScoreTable* (see below).

Player Score Table: Scores for Vehicles, Groups and Static Objects

NOTE:
An unannounced, and probably unwanted change in DCS has rendered the ability to award score for some map object kills (e.g. structures) inactive. You currently cannot attribute kills of many map objects to players. With some luck ED may change this in the future.

Often, there are objects and/or units in your mission that you want to score differently from the default (as explained above). For example, you may have a special reward for the player who manages to find and destroy a special command vehicle hidden somewhere on the map. This vehicle has the same type as many other vehicles of its kind, you just want to award a special score for this individual one. Or you want to award type-specific score that deviates from the default score as explained above: for example, all ground units score 10, but you want to award a score of 25 for any "Avenger"-Type unit destroyed.

This is where the 'Player Score' table comes into play: it allows you to award unit- (or group-) individual scores. A unit that is mentioned in that table has precedence of the default value: whenever a unit is destroyed, Player Score

- first checks if the Player Score table has an entry with **matching name for the unit** (e.g., "SAM Commander") – the name given by a mission designer to a unit. If so, that score is awarded.

- If no match is found, it looks up the unit's **group name, and tries to match** that to an entry in the PlayerScore Table. If a match is found, that score is awarded
- If no match is found, it looks at the **type name of the unit** that was killed (e.g., BTR-80) and tries to find a matching entry in the Player Score table. If a match is found that score is used.
- If no match (name nor type) is found, the matching category (aircraft, helicopter, Ground, Ship, Train) from the default config (see above) is used as score.

Note that the playerScoreTable is used similarly to score destruction of static and scenery/map objects (see below)

To use a score table in your mission, you

- Place a Trigger Zone on the map (anywhere)
- Name that Zone "playerScoreTable" (note: name *must* match exactly)
- Add the names/types and their score to the table.

The playerScoreTable uses the following format:

Name	**Description**
<tape or name of Unit / Group / Static Object> Type Example: BTR-80 Name Example: Big Kahuna	<Score to award 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:



Like units, you can use the playerScoreTable to award a score for the destruction of Static Objects (the ones that you place with Mission Editor, not to confuse with Scenery Objects. For scoring scenery objects, see below).
To have static objects return a score on destruction, simply use the name that you gave it in the "NAME" line of the static's info panel, add it to the playerScoreTable, and set the number of points that you want to award the player who destroys it.

Scenery/Map Objects



Unfortunately, scoring the destruction of scenery objects is often a bit more involved than units and static objects. "Scenery" (map) objects are the many, many map objects that populate the scenery and that come pre-installed on the map.

The big difference between Scenery (map) objects and everything else in DCS is that their name is *not* guaranteed to be unique. While everything else in DCS has a unique name (meaning that there can't ever be two 'things' in DCS that have the exact same name at the same time), this is not true for scenery objects.

This means that if you use a playerScoreTable to award a player some score for the destruction of a map object (which is easy to do), the onus is on you to ensure that there really is no other scenery object with that same name. If there is, players can receive the same score if they happen to destroy a scenery object someplace else that merely happens to have the same name. Now, this is usually not a big issue, but it does open the possibility for some crafty players to increase their score by exploiting this DCS limitation. There is a way to "securely" score the destruction of scenery objects, which I'll discuss soon.



Let's first go the playerScoreTable route of scoring scenery object. All you need to do is add their name and score amount to the table. So how do you get the map object's name? Use the NAME attribute that is shown when you use the 'ASSIGN AS…' right-click function when you click on the scenery object.

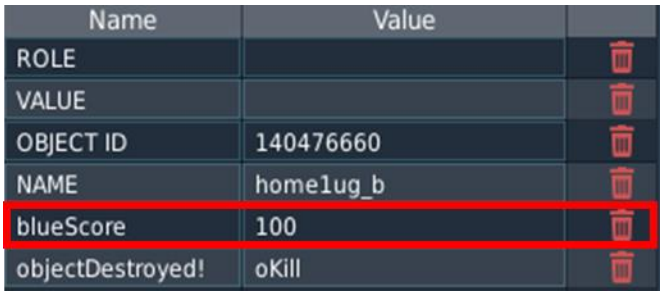


Use what is given as value for the NAME attribute ("most-ferma" in the example to the right). Again, be advised that unlike all other objects in DCS, scenery object names are **not** unique, so make sure that there aren't other objects of the same name close by, or if so, that it is OK to award the score multiple times if same-named objects are destroyed (this often makes sense, as same-name objects are usually of the same shape, for example multiple segments of a bridge).

You can delete the trigger zone that was created over the map object. I prefer to keep it on the map simply to remind myself that I'm using this object's name somewhere. It also helps to transition this solution to the better, uniquely-resolved map object scoring that we'll come to soon.

"Unique" Score for Scenery (Map) Objects

Since scenery objects in DCS behave so differently and do not have unique names, we have a slightly different method to provide "unique" scoring for scenery objects. Use this only if you absolutely must make sure that players may not receive score by destroying other map objects that share the name.



DML has a module that tracks the destruction of map/scenery objects: the awesomely named *objectDestructDetector* (ODD). With it, you can create signals when a specific object on the map is destroyed. PlayerScore provides automatic integration with ODD zones. All you need to do is to add one (or two) attributes to an ODD zone that tells PlayerScore how many points to award a red or blue player for the destruction of that object. PlayerScore and ODD automatically work together

(PlayerScore detects the presence of ODD and works together with it without prompting). The two attributes that link PlayerScore to ODD are

- blueScore – the points awarded to a blue player for destroying the map object
- redScore – the points awarded to a red player

For details, see the objectDestructDetector module (although, truth be told, there's very little else to say)

Award Faction Scores via flag changes

You can award score to a faction simply by changing the corresponding flag. This is because often, you award faction score when the combined actions from players of that faction achieve something and achieving that goal can't (or shouldn't) be attributed to a single player. Examples are bigger operations like capturing an airfield, completing a complex strike etc.

With PlayerScore, all a designer needs to do is to determine if the operation is a success, and when it is, change a flag. PlayerScore then adds the appropriate amount to the faction's score.

Of course, this only works after you tell PlayerScore which flag corresponds to what score and faction. PlayerScore. For this, PlayerScore looks for two specifically named zones in your mission: **redScoreFlags** and **blueScoreFlags**. When it finds any of them, it notes down the information contained therein: which flag to look for, and how much to award that faction when the value of that flag changes.



Name	Value	
killedSAMs	500	

In above example, whenever the mission changes the flag named "killedSAMs", playerScore will add 500 points to the blue coalition's score. There are no restrictions to the number, nor names you can use for flags, I merely recommend that you use speaking names.

You *can* use negative values to reduce a coalition's score

The score listed for a flag is awarded every time that the value of the flag changes.

Feats

Other than kills, PlayerScore also keeps track of 'Feats' that players accomplish, for example successfully completing a CSAR. These feats can be awarded by scripts or other modules. PlayerScore awards a generic 'landing' feat for a successfully landing when the default score for 'landing' is set to a value greater than zero in the config zone.

PlayerScore Feats: placeable with trigger zones and wildcard support
PlayerScore allows mission designers to place feats using zones in ME. Unlike generic feats, these feats support wildcards allowing the mission designer to personalize each feat, make them unique and even limit the number of times that such a feat can be awarded in the course of a mission. When a personalized feat is available, it overrides a generic feat (e.g. landing). Currently, PlayerScore supports the following feats

- *Landing*
  The player who lands inside this feat zone is awarded this feat.

- *Kill*
  The player who kills a unit that is inside this feat zone is awarded this feat

- *PvP*
  Like kill, except that the unit that was killed must have also been controlled by a player for the feat to count

The feats are customized via the feat zone's *description* attribute.

Remember that zones can follow units, so you can easily set up a landing feat zone that follows an aircraft carrier.

The following wildcards are supported in the feat's description attribute:

Feat-specific wildcards

- `<player>` - callsign of the player
- `<kplayer>` - callsign of killed player (or 'unknown AI' when not player-controlled)
- `<punit>` - name of the unit that the player is controlling
- `<unit>` - name of unit that was killed
- `<ptype>` - type (e.g. "A-10A") of the unit that the player is controlling
- `<type>` - type (e.g. "A-10A") of the unit that was killed
- `<pgroup>` - name of the group that the player's unit is with
- `<group>` - name of the group that the unit that was killed belonged to

Standard Wildcards

- `<n>` creates a line feed
- `<z>` - feat zone's name as set with ME
- `<t>` - current mission time in the format "HH:MM:SS", e.g. "08:42:11".
- `<lat>` - latitude of the feat zone's current position
- `<lon>` - longitude of the feat zone's current position
- `<ele>` - elevation of the feat zone's current position.
- `<mgrs>` - feat zone's current position in MGRS coordinates
- `<v: flagName>` - value of flag <flagName>
- `<t: flagName>` - value of flag <flagName> interpreted as time. Format be further configured with the *timeFormat* attribute.
- `<lat: unit/zone>` - latitude of unit/zone with that name
- `<lon: unit/zone>` - longitude of unit/zone with that name
- `<ele: unit/zone>` - elevation of unit/zone with that name.
- `<mgrs: unit/zone>` - mgrs of unit/zone with that name

- `<vel: unit/zone>` - velocity of unit/zone with that name.
- `<alt: unit/zone>` - altitude of unit/zone with that name.
- `<hdg: unit/zone>` - heading of unit/zone with that name
- `<type: unit>` - type (e.g. “A-10A”) of unit with that name
- `<player: unit>` - player callsign who controls a unit with that name

The wildcards follow the general abilities as outlined in the messenger module.

Wildcarded feat Examples:

- `<player> landed on <z> at <t>` for a landing feat could result in
  `Maven landed on USS Enterprise at 02:16:23`
- `<unit> was neutralized by <player> flying a <ptype>` for a kill feat could result in
  `Generalissimo was neutralized by Wet Spike flying a A-10A`

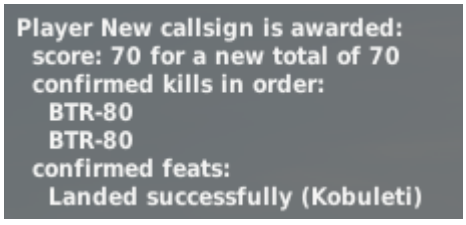
Limit the number of times a feat can be awarded
The personalized feats allow the mission designer to limit the number of times that a feat is awarded to players through the use of the relevant achievement attributes:

- Once per player (`awardOnce`)
- Total number of awards during mission (`awardLimit`)

Deferred awarding of Score / Feats
PlayerScore can withhold awarding scores until the player lands in designated *safeScore* zones. This measure is designed to make players more conscious of their gameplay behavior: they now have an incentive to either try and ‘bring home’ their score versus taking on additional risk.



When this ‘deferred’ option is active, scores accumulate for kills and feats for a player and are awarded upon landing in a zone that is designated ‘scoreSafe’. Once landed, the player must successfully survive for a predetermined time (default 10 seconds) inside that zone. After that, all accumulated score and features are awarded. Should the player lose their aircraft in the meantime (crash, disconnect, switch slot, re-seat same slot), all accumulated scores and feats up to that point are discarded.



WARNING:
Note that deferred feats are discarded when the player fails to claim them. This means that **one-time and limited-amount feats may no longer be available to** players when they

accomplish a one-time feat or limited-amount feat, but do not claim them / are prevented from claiming them.

Score Modifier for Friendly Kills
Friendly fire kills can be scored differently than normal kills: they are multiplied with a configurable value (the 'ffMod' attribute). At best, you should set that value is 0 (so killing a friendly unit does not earn any score), and it can also be set to a negative value to emphasize the seriousness of the error. By default, that value is set to -2 (negative two).

Score Modifier for PvP kills
Similar to friendly kills, the score for a player kill is multiplied by a factor (settable with the 'pkMod' attribute). By default this value is 1 (same amount as AI kill), but you could reward hunting enemy players by setting this value higher (for example 2). Note that in case of accidentally killing a friendly player, the factors are cumulative: if pkMod is 10 and ffMod is set to -4, the resulting factor is -40, which can serve as a great disincentive to ever do that again.

Score for Players Losing their Aircraft
PlayerScore can award points (usually a negative amount) when a player loses their airframe. By default, this amount is set to zero (no consequences). If you set this amount to anything other than zero (make it a negative value to not reward plane loss), that amount is awarded *immediately* (for obvious reasons: no deferred scoring after landing is possible).

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.

Getting Current Score


The PlayerScore module itself does not interact with players, it merely collects kills and feats, and can save them to disk upon request. A second, tiny and highly efficient module, *cfxPlayerScoreUI* installs a menu for all player units that allows them to request their current score. PlayerScoreUI requires no configuration, it simply works as soon as you add it to the mission.

Export Score Lists/Rankings as plain text
PlayerScore can export the current score list (all players recorded) as a plain text file. The list can be ranked (by score), and it can be exported incrementally (i.e. it will append player score snap shots to an existing file, create a new one otherwise).

Mission designers can specify a file name for the text file that is to be created. Note that in order to export player scores to file, the mission must also include the *persistence* module and the DCS installation must be de-sanitized.

Automatic integration with other modules
PlayerScore automatically provides integration with the following modules:

- PlayerScoreUI (shows requesting player's score)
- ObjectDestructDetector (unique map object score)
- CSARManager (feats for successfully evacuating downed pilots)
- HeloTroops / Owned Zones (feats for dropping troops into non-aligned owned zones)
- WilliePete (killing units through artillery after marking targets)

5.4.1.2 Dependencies
PlayerScore requires the modules dcsCommon and cfxZones.

If you want to save the score table to storage, you must also include the module *persistence*.

PlayerScoreUI requires the PlayerScore module.

5.4.1.3 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 anywhere on the map in ME
- 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
pkMod	Factor to multiply score for a player-kill (PvP). Defaults to 1 (same score)
ffMod	Factor to multiply score with for a friendly kill. Set to 0 to award no

Name	Description
	points for friendly kills. Defaults to -2 (negative double score for friendly fire kills)
planeLoss	Score to award when a player loses their airframe. This score is warded immediately. Set i to a negative value to deduct score from a player. Defaults to 0 (zero, plane loss has no consequence)
landing	Score for a successful landing, will also award a landing feat. Is only active if the value for landing is greater than zero. Also applies to helicopters. Default: 0 (no score, no feat awarded for landings).
announcer	If false, no kills are announced. Score is still kept. Defaults to true
scoreSound	Name of the sound file to play when a score is announced
badSound	Name of the sound file to play when killing own troops or being killed in PvP
saveScore?	Watchflag. When triggered, PlayerScrore saves the current score table to a plain text file. Requires the 'persistence' module be active in the mission, and the hosting DCS installation be de-sanitized. Defaults to <none>
incremental	When set to true, score save to plain text (via saveScore?) are appended to the save file if it already exists. Defaults to false (overwrite existing player score file)
rankPlayers	When set to true, players are ranked by their achieved score when saving player score to storage Defaults to false
scoreOnly	When set to true, a player's score omits all feat details on export to plain text file Defaults to true
deferred	When set to true, scores and feats are only awarded after the player successfully lands and remains for a certain time in a designated 'safeScore' zone. Defaults to false (immediately awards score)
delayAfterLanding	Number of seconds a player must remain in a safeScore zone after a successful landing before they are rewarded with their accumulated score. Should they die in the interim, all accumulated scores are discarded. Only applicable if deferred is true. Default: 10 (seconds)
scoreFileName	Name of the plain text file (without extension) that scores are exported to when saveScore? is triggered. Defaults to "Player Scores"
reportScore	When queried to report the player's statistics, should the module include score? Set to false to suppress the player's score. Defaults to true (score is reported)
reportFeats	When set to true, the module also reports feats when queried for the player's statistics. Deafults to true (feats are reported)
reportCoalition	When set to true, the module reports the coalition's total with a player's score. Defaults to false (no coalition score reported)
noGrief	Controls if negative player scores also affect the player's coalition score. By default, to prevent 'griefing' (i.e. childish behavior), when a player is awarded demerits (negative score), that is only subtracted

Name	Description
	from the player's total, but not the coalition. That way, a griefing player can't intentionally sabotage a coalition's score by committing fratricide or crashing planes. When set to false, this protection is disabled, and griefing would be possible. Defaults to true (griefing protection is on, only positive player scores affect coalition score)
redScore#	Immediate Output flag that always contains the current total score for RED faction Defaults to <none>
blueScore#	Immediate Output flag that always contains the current total score for BLUE faction Defaults to <none>
score2finance	A conversion factor to convert score points to finance 'points' (funds) with the *bank* module. Factor applies to all score points that are awarded to a coalition. Defaults to 1.0 (score points are passed straight as fund to bank)

PlayerScoreTable

To use a score table in your mission, you

- Place a Trigger Zone on the map (anywhere)
- Name that Zone "playerScoreTable" (note: name *must* match exactly)
- Add the names/types and their score to the table

The playerScoreTable uses the following format:

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

5.4.1.4 ME Attributes

Score Safe (deferred scoring)
When you set the *deferred* = *true* attribute in the config zone (i.e. scores are only awarded after landing in a scoreSafe zone) you must designate zones where players can land in before they can be awarded a score. If you enable deferred scoring and have no scoreSafe zones designated, playerScore will give a warning on mission start.

Name	Description
scoreSafe	Designates this entire zone as a zone where players can land inside and, after successfully landing and remaining the required amount of time (see *delayAfterLanding* config attribute) inside the zone, have their accumulated score and feats awarded. The value of this attribute determines which faction can have their scores awarded after landing as follows: • 0 or 'neutral': all factions • 1 or 'red': only red faction • 2 or 'blue': only blue faction Defaults to 'neutral' (both red and blue can land here to have their scores awarded) **Mandatory**

Player-defined Feats (Kill / Landing) (Optional)
Mission designers can place zones in which players can achieve feats. To place a feat, add a trigger zone, and then add the following attributes to the zone

Name	Description
feat	Designates this zone as a zone in which the completion of certain actions leads to a feat being awarded to the player who completes it. The value of this attribute can restrict the feat to certain factions: • 0 or `neutral` – all players can achieve this feat • 1 or `red` – only red players can achieve this feat • 2 or `blue` – only blue players can achieve this feat Defaults to 0 (all players can achieve this feat) **Mandatory**
featType	What a player needs to accomplish in order to have this feat awarded. Currently supports the following • `land` or `landing` – player lands their unit inside this zone • `kill` – players kills a unit that is currently inside this zone • `pvp` – player kills a unit that is currently inside this zone and is controlled by another player. Defaults to `kill`
description	Text of the feature that is to be awarded to the player. The description supports **wildcards** as follows. Feat-specific Wildcards: • `<player>` - callsign of the player • `<kplayer>` - callsign of other player when that plane was controlled by a player, 'unknown AI' else. • `<punit>` - name of the unit that the player is controlling • `<unit>` - name of unit that was killed • `<ptype>` - type (e.g. "A-10A") of the unit that the player is controlling • `<type>` - type (e.g. "A-10A") of the unit that was killed • `<pgroup>` - name of the group that the player's unit is with

	• <group> - name of the group that the unit that was killed belonged to General Wildcards • <n> creates a line feed • <z> - feat zone's name as set with ME • <t> - current mission time in the format "HH:MM:SS", e.g. "08:42:11". • <lat> - latitude of the feat zone's current position • <lon> - longitude of the feat zone's current position • <ele> - elevation of the feat zone's current position. • <mgrs> - feat zone's current position in MGRS coordinates • <v: flagName> - value of flag <flagName> • <t: flagName> - value of flag <flagName> interpreted as time. Format be further configured with the *timeFormat* attribute. • <lat: unit/zone> - latitude of unit/zone with that name • <lon: unit/zone> - longitude of unit/zone with that name • <ele: unit/zone> - elevation of unit/zone with that name. • <mgrs: unit/zone> - mgrs of unit/zone with that name • <vel: unit/zone> - velocity of unit/zone with that name. • <alt: unit/zone> - altitude of unit/zone with that name. • <hdg: unit/zone> - heading of unit/zone with that name • <type: unit> - type (e.g. "A-10A") of unit with that name • <player: unit> - player callsign who controls a unit with that name Defaults to (some feat)
awardLimit	Total number of times that this feat can be awarded during the life time of a mission. A negative value means 'infinite number of times' Defaults to -1 (feat can be accomplished an infinite number of times)
awardOnce	If set to true, this feat can only be awarded once per player. Note that awardLimit and awardOnce work cumulative: if you set awardOnce to true, and awardLimit to 2, the feat can be accomplished twice, but only by different players. Defaults to 'false', a player can achieve this feat multiple times during the same mission

Kill Zones (optional)

The presence of kill zones automatically switches PlayerScore to awarding kill score and kill feats exclusively to kills that happen entirely inside a kill zone (i.e. killing unit and target are inside the same kill zone at the time of the kill).

Name	**Description**
killZone	Designates this zone as a zone in which killing other units can earn score or feats. All kills outside this or other kill zones do not award a score or feat. To count, at least the target must be inside a kill zone, optionally (see *duet* attribute, below) you can also enforce that the victorious player must be inside the same kill zone.

	The value of this attribute is ignored **Mandatory**
duet	If set to true, *both* units (player and target) must be inside the *same* kill zone. Defaults to 'false' (only killed unit must be inside kill zone)

Flags to trigger coalition score
You can use the specially named zones "blueScoreFlags" (for blue coalition) and "redScoreFlags" (red coalition) to award score whenever one of the flags listed as attribute name changes its value. Spelling of the zone's name must match exactly.

Name	**Description**
<any flag name>	Whenever the value of the flag listed here changes, PlayerScore adds the amount given in the Value field to the relevant coalition. Negative values are valid. If you give a value of 0 (zero) or non-numeric value, PlayerScore will give you a warning when it starts the mission. The value of this attribute is ignored

5.4.1.5 Persistence
PlayerScore is persisted together with each player's kill and feats list.

Adding the persistence module also enables export of a plain text score to a file of your choosing.

5.4.1.6 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.

Add zones named "blueScoreFlags" or "blueScoreFlags" (optionally) to be able to add score to a coalition by use of flag changes.

Optionally, add the PlayerScoreUI module to allow players to see their own score on demand.

5.4.2 cfxHeloTroops (Troop Transport)



5.4.2.1 Description

HeloTroops adds the ability for players to Airlift (transport: load / unload) ground troops with transport airframes to missions. By default, this is available only to DCS's troop transport helicopters (Huey, Hip, Hook and Hind), and it can be easily changed (via a config zone) to extend this capability to any aircraft (including fixed wing).

The module installs an "Airlift Troops…" command into an eligible player's Communication→**F10 Other**… menu to allow them to load and unload troops, and to set troop transport preferences.

NOTE:
Alert your players to **not confuse** this with DCS's own "Airborne Troops" that is available to many helicopters.

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).

When flying an eligible transport airframe, the script loads and deploys troops *when the aircraft is on the ground*. HeloTroops has special built-in abilities to better support mission design:

- HeloTroops' **dropZones** can restrict a player to unload troops only in designated drop zones.
- HeloTroop's dropZones can send signals when a player successfully disembarks troops inside a dropZone
- HeloTroops provides mechanisms to automatically enmbark and disembark troops. This ability can be controlled by players.
- When close to a **cfxSpawnZone** with a 'requestable' attribute, players can trigger a spawn via the communications menu.

- Landing close to a group that entirely consists of transportable troops (as defined in HeloTroops), allows the player to load these troops into the helicopter for transport.

As mentioned above, if a player aircraft lands successfully with troops loaded, these troops can be (automatically) deployed. HeloTroops supports player-configurable settings to auto-load the closest loadable group when no troops are 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*. This 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 other 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)
- **GroupTracker** – Groups that are being picked up (and are therefore temporarily removed from the mission) will still be tracked by GroupTracker and do not cause a ‘remove’ nor ‘add’ event.
- **CSAR Manager** – compatible with UI and loading of downed pilots.

Other
HeloTroops fully supports multiplayer. In MP missions, player groups **must** be single-unit groups or the scripts will not work correctly.

Currently, the script does not manage the aircraft’s cargo weight when units embark or disembark. This feature me be added in the future.

5.4.2.2 Dependencies
Required: dcsCommon, cfxZones, commander, groundTroops

Optional: cfxSpawnZones (for requestable troop spawning)

5.4.2.3 ME Attributes
HeloTroops can use *dropZones* that serve two purposes:

- Restrict troop disembarkment (unloading) to inside these zones. This requires the configZone's enforceDropZones attribute to be set to true
- Generate output signals when a player disembarks troops inside this zone

Name	Description
dropZone!	Tells DML that this zone is a zone with special relevance for HeloTroops. The value of this lists the flags (commands) to signal if a player-controlled helicopter disembarks troops inside this zone. There is an option in heloTroop's config zone to enforce drop zones. If enabled, players can only disembark troops when they land inside drop zones. This option is **off by default**. **MANDATORY**
dropMethod	DML method used to send the signal over outputs Defaults to "inc"
coalition	Colation that can drop here. If set to neutral, both coalition (red and blue) can drop here Defaults to 0 (neutral) – both coalitions can disembark their troops here
autoDespawn	Time interval in seconds after which disembarked troops despawn. Used primarily to clear the drop zone in extraction-based missions that would overcrowd otherwise. Set to a value less than 1 to turn off Defaults to -1 (no auto-despawn)

5.4.2.4 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 list (comma separated) 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"
troopCarriers	A list of helicopter types that are allowed to carry troops in this mission. Defaults to DCS Common's list of troops (which is usually Mi-8MT, UH-1H, Mi-24P, OH58D, CH-47Fbl1), but you can provide your own list (for example to add non-official types). Example: "Mi-8MT, UH-1H, SA342Minigun, C-130J" removes the Hind and adds the Gazelle in Minigun configuration as well as the Hercules fixed-wing transport to the list of legal troop carriers. Supports wildcard type endings: if a type ends on an asterisk ("*") all types that match whatever precedes the asterisk are accepted. For example, "Mi-*" will match both "Mi-8T" and "Mi-24P". You can supply the type 'any' or 'all' to allow all player aircraft to carry troops. You can supply the type "helo" or "helos" to allow all helicopters to carry troops Default <none> (use dcsCommon's list of troop carriers)
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 a helicopter. Defaults to 100 meters
combatDropScore	Score to award when a player unloads troops in a non-aligned owned zone. Requires OwnedZones and PlayerScore to be active. Defaults to 200
actionSound	Name of the sound file to play to the helicopter group whenever troops are loaded or unloaded. Can be overridden by 'loadSound' and 'disembarkSound' Defaults to <none>
loadSound	Name of the sound file to play to the helicopter group whenever troops are loaded into the helicopter. Overrides 'actionSound' Defaults to the value set by actionSound (above).
disembarkSound	Name of the sound file to play to the helicopter group whenever troops are unloaded from the helicopter. Override 'actionSound'. Derfaults to the value set by 'actionSound' (above).
requestRange	Distance (in meters) that a player helicopter must land within for a spawner or cloner with 'requestable' attribute to be eligible to request troops from. Note that this distance is different (usually greater) that the pick-up range (see above), so helicopters can order a spawner/cloner to produce fresh troops, but may not be close enough to pick them up (i.e., the player must move the closer to pick them up). Defaults to 500 (meters)
enforceDropZones	When set to true, a player can only disembark troops when they land inside a *dropZone,* Defaults to false (players can disembark their troops anywhere)

5.4.2.5 Persistence

Helo Troops keeps track of the units it deploys (after picking them up), and they will appear correctly after loading a mission, with their orders intact.

5.4.2.6 Using the module

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

Add dropZones as you see fit

5.4.3 csarManager

5.4.3.1 Description

csarManager is a drop-in module for DCS Missions that adds CSAR (Combat Search And Rescue) capabilities to missions. It allows CSAR missions to be created from CSAR trigger zones while the mission is running (i.e. dynamically) and integrates with other modules that can also automatically create CSAR missions (e.g., autoCSAR and limitedAirframes). By default, CSAR missions are only available to troop transport helicopters (i.e., Huey, Hind, Hip – but not Shark, Gazelle, Kiowa nor Apache), and this can be changed with a simple attribute in csarManager's config zone.



MISSION CREATION / RANDOMIZATION

When triggered, CSAR zones dynamically create the required units on the ground (or in water) and make the corresponding mission available for eligible players. You control the location of the evacuees (with the trigger zone), if the spawned unit(s) also broadcast an emergency signal so potential rescue helicopters can home in on to their location with **ADF**, if they signal via smoke or flares and more.

To allow for randomization, the evacuee that is to be rescued is placed at a random location inside the trigger zone (this can be turned off). This means that a small trigger zone will result in a very precise placement of the unit, while a large trigger zone means that the evacuee is placed anywhere inside that zone. Furthermore, you can – to simplify simulating traffic accidents – tell CSAR manager to place evacuees on the nearest road (you may want to disable local traffic in this case), or, when dealing with large, built-up areas like for example on the Syria and Sinai map, even tell it to place the evacuee inside those warrens (which can generate terrifyingly difficult missions).

APPROACH AND VISUAL CUES

When the player closes in on the evacuee, they are alerted to that fact, and the direction to the evacuee is given in “o’clock” format (as in “check your 3 o’clock”). When enabled (as per default), the evacuee’s location is marked by blue smoke, and a (also configurable) by a red flare (helpful at night).

MISSION COMPLETE

For CSAR missions to be completed for that coalition, it must have at least one CSARBASE zone. These zones define where a rescue helicopter from that coalition can drop off rescued personnel – a CSAR reception center (hospital or debrief center). Without a CSARBASE zone, helicopters can pick up and drop off downed pilots, but lacking a completion zone, the CSAR missions cannot complete. When a player lands in a CSARBASE zone, all currently loaded rescued troops are automatically unloaded, and the mission is completed. If the

PlayerScore module is also available, the player who controls the helicopter also receives the reward/feats set forth by the mission.

OTHER FEATURES

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 proximity or (over open water, steep inclines or forest) 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.

SINGLE-UNIT PLAYER GROUPS

Due to some DCS shortcomings, CSARmanager currently requires that all player helicopters that are to perform CSAR missions be placed in single-unit groups.

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 a signal is received on the startCSAR? input

Name	Description
in? startCSAR?	Watches this input for a DML signal. **Each time that the input triggers, a new CSAR Mission is created.** Defaults to <none>
triggerMethod	The DML method to trigger inputs. Defaults to "change"
rndLoc	If set to false, the evacuee is placed in the center of the CSAR zone, else in a random location inside the CSAR zone. Note that CSAR manager *does* support polygonal zones Defaults to **true** (randomized location inside CSAR zone)
onRoad onRoads	If set to true, the evacuee is always placed on the closest point on a road next to the intended spawn point. Note that this can result in evacuees spawn outside the CSAR Zone. Defaults to false (no road spawn enforced)

INTEGRATION WITH LIMITED AIRFRAMES

The limited airframes module integrates with CSAR Manager to automatically generate a CSAR mission when a player pilot safely ejects from a plane or ditches it outside a designated airfield/FARP/ship. To correctly work together, **CSAR Manager must load before Limited Airframes**.

INTEGRATION WITH PLAYERSCORE
When you use the PlayerScore module, each successful CSAR mission is listed as an individual feat, and can add points to a player's total. You can control the number of points earned for a scar on a per-mission base via the "score" attribute in a "CSAR" mission zone, and via the "rescueScore" attribute in the "csarManagerConfig" zone. Individual scores override the global configuration setting.

INTEGRATION WITH AUTOCSAR
csarManager automatically integrates with autoCSAR (a module that automatically creates CSAR mission when a pilot ejects).

INTEGRATION WITH SCRIBE
csarManager automatically integrates with the "scribe" logbook module. Whenever a player successfully rescues someone, the "rescued" logbook counter is increased.

INTEGRATION WITH RADIOMENUS
Usually, csarManager installs its UI at the communications→F-10 Other level. If you are using radioMenus' cascading feature, you can instead direct csarManager to attach the csarManager UI to any triggerZone with a radioMainMenu attribute.

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

Optional: commander, limitedAirframes, autoCSAR, names, radioMenus

5.4.3.3 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 performance impact when close in a helicopter, so be mindful of this option Defaults to true (activate smoke)
smokeColor	Color of the smoke to pop when helicopter is in range and enabled. Understands numbers (0-4) and names (green, red, white, orange or blue) Defaults to "blue"
useFlare	When a player approaches the evacuees, release a flare. This is especially helpful in night-time recovery. The flare is released 5-10

	seconds after the player comes into range Defaults to true (launch a flare)
flareColor	When flares enabled, the color of the flare that is launched. Possible colors are -1, “random” or “rnd” for random, or 0 (zero) to 3 or one of “green”, “red”, “white”, or “yellow” Defaults to “red”
csarRedDelivered!	Flag to change when a red coalition-flown CSAR mission ends successfully
csarBlueDelivered!	Flag to change when a blue coalition-flown CSAR mission ends successfully
csarDelivered!	Flag to change when a CSAR mission ends successfully
rescueRadius	A player-controlled helicopter must land within this distance (in meters) to the target to pick up. Defaults to 70 meters
hoverRadius	When attempting a hover rescue, helicopter must stay within this range (in meters). Defaults to 30 meters
hoverAlt	When attempting a hover rescue, helicopter must stay below this altitude (in meters). Defaults to 40 meters
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 Defaults to 2000 meters
beaconSound	Name of sound file (ogg or wav) to play on the ELT frequency. Includes extension. Example: "Radio_beacon_of_distress.ogg"
pilotWeight	Weight for an evacuee in kg. Defaults to 120kg
hoverDuration	Time required to hover above pilot to secure winch and complete rescue Defaults to 20 seconds
rescueScore	Default number of points awarded for a successful rescue. Awarded upon delivery in CSARBASE. Requires PlayerScore module to have effect Defaults to 100
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) Defaults to true (vectoring enabled)
actionSound	Name of sound file to play on notifications
successSound	Name of sound file to play on successful mission completion. Defaults to whatever you have defined as *actionSound*.
addPrefix	Add “downed “ before the mission name. Pure eye candy. Defaults to “true” (add eye candy) “downed” to mission name
useRanks	Add a random military rank before the name, e.g ‘Lt”, “Col” or “WO”. The ranks are taken from the list that you can customize with the ‘ranks’ attribute (see below) Defaults to “false” (no ranks added)
ranks	A list (separated by commas ‘,’) of military ranks that is added in front of a name if the useRanks option is true. You don’t have to add ‘real’ ranks, merely a list of words of which one is picked and prepended before the name. If you add a word multiple times, it becomes more likely to be chosen than others.

	Defaults to "`Lt, Lt, Lt, Col, Cpt, WO, WO`" – the rank "Lt" is three times as likely to be chosen, "WO" twice as likely as "Col" or "Cpt".
troopCarriers	A list of helicopter types that are allowed to carry/rescue troops in this mission. Defaults to DCS Common's list of troop carriers (which is usually `Mi-8MT, UH-1H, Mi-24P`), but you can provide your own list (for example to add non-official types). Example: "`Mi-8MT, UH-1H, SA342Minigun`" removes the Hind and adds the Gazelle in Minigun configuration to the list of legal troop carriers. Supports wildcard type endings: if a type ends on an asterisk ("*") all types that match whatever precedes the asterisk are accepted. For example, "`Mi-*`" will match both "`Mi-8T`" and "`Mi-24P`". You can supply the type '`helos`' to allow all player helicopters (including unofficial Mods like Blackhawk) to carry troops. You can supply the type '`any`' or '`all`' to allow all player aircraft to carry troops. Default <none> (use dcsCommon's list of troop carriers)
maxMissions	The maximum number of missions that are listed for players when querying open missions. Defaults to 15
comsRange	The maximum range (in meters) at which it is broadcast that a mission has expired, measured from the location of the evacuee Defaults to 40000 (40 km)
rescueTypes	A list of DCS types, separated by comma (",") from which csarManager picks one each time that it creates an evacuee. May become useful once DCS provides pilot and other units to pick from, for the time being the default is good enough. Defaults to "`Soldier M4 GRG`"
timeLimit	If given, a range (e.g "60-120") of **minutes** in which the pilot must be picked up. If only one number is given, that number will be set for all pilots. In the above example, a random value between 60 minutes (one hour) and 120 minutes (2 hours) is selected. If no timeLimit attribute is given in the config zone, no global time limit is applied to a csarMission If a CSAR zone specifies its own time limit, that zone's time limit overrides this global's setting. Defaults to <none> (no global time limit for csar missions)
attachTo:	When present, csarManager's UI attaches to the trigger zone named as value. This trigger zone must have a "radioMainMenu" attribute. Note that if you use this attribute, you must also add the radioMenus module to your mission and that module **must** load before csarManager. Defaults to <none> (UI installs at F10 Other level)

5.4.3.4 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.

5.4.3.4.1 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 that lands inside this zone completes CSAR missions for those pilots that it is carrying. 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 cannot be completed. There is no upper limit on the number of CSAR Bases a side can have. The value of this attribute defines which sides can complete a CSAR mission here. • "red" or 1 means that only red pilots can complete their missions here, • "blue" or 2 means blue only, and • "neutral" or "0" means that all sides can complete CSAR missions in this zone. Defaults to "neutral" – all sides can complete CSAR Missions in this zone. **MANDATORY**
name	Optional name for CSARBASE. Defaults to trigger zone's name

5.4.3.4.2 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. The unit is placed anywhere within the trigger zone, so the size of the trigger zone determines the degree of randomness (currently, only circular trigger zones are supported).

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. Contains the name of this mission, recommended is to use a personal name, e.g. "Lt. Wesley Crasher" Names can be randomized by setting the name to "*rnd". In this case,

	each mission has a random name (first and last name). Using this option requires the "**names**" module to be present. **MANDATORY**
coalition	Faction (red/blue) for which this mission is generated Defaults to NEUTRAL
freq	Frequency for the ELT (radio to home in on) in KHz. Defaults to Random
timeLimit	If set, the number of **minutes** from the point of CSAR mission creation until the evacuee is considered dead or missing. Players have time to pick up the player before the time limit runs out. If no time limit is given, the evacuee is considered stable and the mission will not expire You can enter a range (e.g. "45-120") to randomize the time limit between those numbers. Defaults to none (no time limit)
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? start? startCSAR?	When this input triggers, a new CSAR is created based on this Zone's attributes. Defaults to <none>
score	Number of points to award when rescued successfully. Only relevant when PlayerScore is installed. If not present, csarManager's default score is used.
triggerMethod	DML Method that triggers inputs. Defaults to "change"
rndLoc	If set to false, the evacuee spawns at the CSAR Zone's center, or at a random location inside the CSAR zone otherwise. Note that CSAR Zones support polygonal zone shapes Defaults to TRUE (random spawn location inside the zone)
onRoad onRoads	If set to true, the evacuee spawns on the nearest road closest to the intended (even when randomized) spawn location. Note that this can cause evacuees outside of the CSAR zone. Defaults to false (units are not moved to the nearest road)
clearance inBuiltup	If given, the units are placed on the map with at least as much clearance (in meters) to the center of any map objects in the vicinity. Use this when you want to spawn units on a free spot in map regions that have lots of houses like cities (hence the 'builtUp' name). **Note:** There is no guarantee that DML can find a sufficiently open (free) space inside the destination trigger zone. For example, some densely populated areas on the "Syria" or "Sinai" map are so densely packed with objects that it is impossible to quickly find a free space that is 20 meters wide. After several tries, it simply gives up and places the unit randomly without guarantee that the inBuiltup constraint (clearance) is kept. **Recommendation**: use 15 (meters = 45 feet). If that works well, you can try and decrease or increase, depending on your needs. Is overridden if *onRoad* is set to true.

	Defaults to <none>

5.4.3.5 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

5.4.4 autoCSAR

5.4.4.1 Description

AutoCSAR provides automatic CSAR mission generation for any pilots (human and AI) that successfully eject from their plane. It builds on, and therefore requires, CSARManager.



Missions with capturable zones / airfields may incorrectly react to CSAR missions created from ejected pilots landing too close to them. To avoid this and similar exploits, you can use the 'noExploit' attribute

5.4.4.2 Dependencies

autoCSAR requires dcsCommon, cfxZones and csarManager.

5.4.4.3 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 anywhere in ME
- Name it "autoCSARConfig" (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"
red	If this attribute is true, CSAR missions are automatically created for red pilots Defaults to true
blue	If this attribute is true, CSAR missions are automatically created for blue pilots Defaults to true
noExploit	If true, ejected pilots generate no CSAR mission when within a certain range (see killDist, below) to airfields or FARP

Name	Description
	Defaults to false (pilots landing within killDist to an airfield or FRAP do generate a CSAR mission)
killDist	Maximum Distance to an airfield or FARP which will deny a new CSAR mission when noExploit is true. Defaults to 2100 (meters)

5.4.4.4 ME Attributes

None

5.4.4.5 Using the module

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

5.4.5 Limited Airframes

5.4.5.1 Description



Limited airframes is a drop-in module that works out-of-the-box to put a limit on the number of player-controlled pilots per side (slightly misleading module name notwithstanding). When that number drops below zero, that side loses the engagement (limited airframes will not end the mission, merely set a flag accordingly). This feature can be turned on and off in-mission, and each side can have a different maximum.

Limited Aircraft is fully multiplayer capable and comes with a comprehensive in-game menu.

HOW TO LOSE A PILOT

Limited Airframes monitors world events and looks for player events that indicate that a player's plane is lost (hence the module name). When limited airframes determines that the airframe is lost, it lowers the number of player pilots for that side by one.

Limited Airframes deducts a pilot when

- The player is killed via crash or other means
- The player ejects
- The player's aircraft is ditched (player exits the plane outside designated safe zones)
- The player changes aircrafts while not landed (even if inside a pilot safe zone)

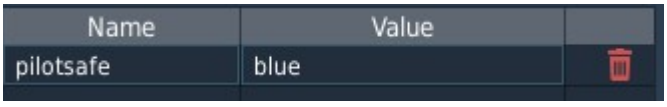
Note that other modules (e.g., CSAR Manager) can work with Limited Airframes to automatically generate missions to rescue downed pilots and replenish the number of remaining pilots.

HOW TO *NOT* LOSE A PILOT (SAFE ZONES)

Limited airframes will not deduct a pilot when

- Player changes airframes inside a designated safe zone with the aircraft safely landed.

Safe zones are designated by a trigger zone with the 'pilotsafe' attribute. Safe zones can be safe for any coalition (default), or blue and red separate.



Since DML internally supports zone ownership, a zone that is owned by one side will not be safe for the other side to change airframes in. Pilotsafe zones owned be neutral factions are safe.

HOW TO RE-GAIN A PILOT

Limited airframes provides an API to increase the number of available pilots per side. Other modules use this API (e.g. CSAR Manager). Note that even using the API it is not possible to increase the current number above the imposed maximum.

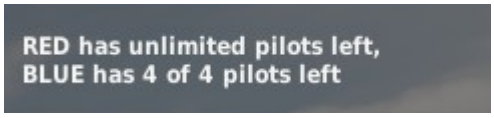
ME INTEGRATION

Limited Airframes provides support for DML and other flags for the following events:

- Red won (blue ran out of pilots) – DML Flag
- Blue won (red ran out of pilots) – DML Flag
- Number of pilots left blue
- Number of pilots left red

UI

Limited Airframes has built-in UI via the Communication→Other game menu. The Ui allows each side to enquire the number of pilots they have left, and turn limited airframe count on/off.

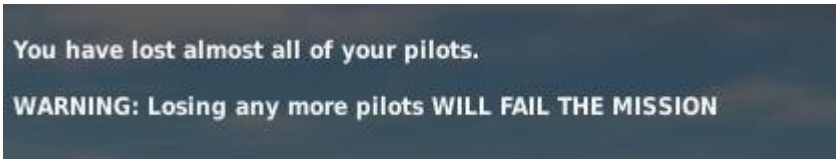


Allowing players control over airframe limits can be disabled with the config zone's '`userCanToggle`' attribute. When disabled, players can only inquire the current status (remaining aircraft pers side). The entire UI can be disabled with the config zone's '`hasUI`' attribute.

Each time a side loses a pilot, a message is displayed with announces the number of pilots remaining for that side.

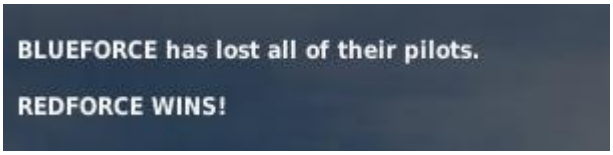


When a side is about to lose because it is running out of pilots, a warning appears for that side to that effect



Once a side has lost all pilots and loses the engagement, another message comes up:

All above messages can be accompanied by audio. The file names to play are set with the config zone.



SETTING LIMITS

Use the module's config zone to set up the upper limits for pilots for each side. Supply "-1" as maximum to allow an unlimited supply of pilots for that side.

LIMITATIONS
(Handling in case of multi-seat planes)

5.4.5.2 Dependencies
limitedAirframes requires dcsCommon, ~~cfxPlayer~~, cfxZones

If you want limited Airframes to automatically interact with the CSAR Manager module, **CSAR Manager must load before Limited Airframes**, or limited Airframes will not connect to CSAR Manager.

5.4.5.3 Module Configuration
Limited Airframes uses a configuration zone for setting main options. To configure this module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it “limitedAirframesConfig” (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”
enabled	Controls whether Limited Airfames is active. Defaults to “true”
userCanToggle	Controls whether players can turn Limited Airframes on and off during the mission. Defaults to “true”
hasUI	Controls whether players have access to Limited Airframes UI via Communications→Other. Defaults to true (users have access to UI)
maxRed	Maximum (and starting) number of pilots for the red coalition. Set to -1 to make the number unlimited. Defaults to -1 (unlimited)
maxBlue	Maximum (and starting) number of pilots for the red coalition. Set to -1 to make the number unlimited. Defaults to -1 (unlimited)
red# ~~#red~~	Flag that continuously is set to the current number of pilots remaining for Red
blue# ~~#blue~~	Flag that continuously is set to the current number of pilots remaining for blue
redWins! redWinsFlag!	Flag to bang! when blue has lost all pilots and red wins. Defaults to <none>
blueWins! blueWinsFlag!	Flag to bang! when red has lost all pilots and blue wins. Defaults to <none>
method	DML bang! method. Defaults to ‘inc’
warningSound	Name of sound file to play when limited airframes is displaying a message. Defaults to <none>
winSound	Name of sound file to play for winning side when other side has lost all pilots. Defaults to <none>
loseSound	Name of sound file to play for the side that has lost all pilots and therefore lost the engagement. Defaults to <none>
announcer	When set to false, the are no announcements for change on air frames, Defaults to true (airframe changes are announced)

5.4.5.4 ME Attributes

Limited Airframes uses ME trigger zones to denote areas where player pilots can safely change into other airframes without losing a pilot. These zones can have the following attributes:

Name	Description
pilotsafe	Marks this zone as safe for pilots to change into other airframes when landed. • If the value to this attribute contains neither the word 'red' nor 'blue', it is safe for all coalitions • If the value of this attribute contains the word 'red' this zone is safe for the red coalition • If the value of this attribute contains the word 'blue' this zone is safe for the blue coalition Note that safe state may be contingent on ownership of the zone. **MANDATORY**

5.4.5.5 Persistence

Limit Airframes persists the number of pilots left for each side

5.4.5.6 Using the module

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

In ME, place at least one zone that has a "pilotsafe" attribute when that side has a non-negative (i.e. finite) number of pilots.

5.4.6 Guardian Angel

5.4.6.1 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 aircraft 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!)
- Can be turned on and off at will with flags
- Also avoids newer re-targeting missiles like S-300
- Supports cloned units

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).

HOW TO TELL GA WHICH AIRCRAFT TO PROTECT

By default (which you can modify with a config zone), Guardian Angel protects the following aircraft

- All player aircraft (all coalitions). This can be changed with the 'autoAddPlayer' attribute (see Module Configuration, below)

This means that all player aircraft are protected, while all AI aircraft are unprotected.



In order to **individually** protect aircraft (including AI), or remove protection, place these aircraft in trigger zones with a 'guardian' attribute, and either add true (aircraft is protected) or false (not protected) as value. These 'guardian' zones override all config settings, so when you place player aircraft

inside a trigger zone with <gtellsian = false> attribute, that plane will not be protected even if the configuration tells Guardian Angel that all player aircraft should be protected.

Name	Description
guardian	**MANDATORY** Tells Guardian Angel how to treat aircraft inside the trigger zone: • *true* All aircraft inside this zone are protected by Guardian Angel • *false* All aircraft inside this zone will not receive protection from Guardian Angel. Defaults to 'true' (all aircraft inside are protected)

LIMITATIONS

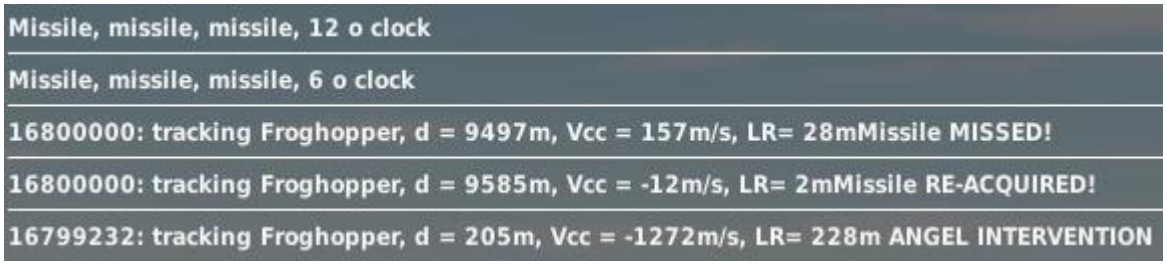
Guardian Angel works by tracking all missiles fired at aircraft, calculating their closing velocity to the targeted aircraft, and deriving a lethal distance. When inside lethal range, the missile is intercepted for protected aircraft. That way, fast closing missiles are intercepted further away than slowly closing ones.

Even though the predictive code is highly successful at intercepting missiles aimed at a protected aircraft, they can still be destroyed through collateral damage caused by an explosion nearby:

- When another plane that happens to be too close to the protected plane is destroyed and the blast kills the protected aircraft
- When a missile that targeted a different plane loses track and explodes in close proximity
- When a missile tracks countermeasures (decoys, flares, chaff) and as a result explodes near the aircraft.

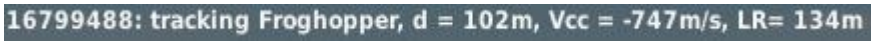
WARNINGS

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.

Turning Guardian Angel On/Off at will during the mission
Guardian Angel permanently watches two flags that you can specify to turn it on and off. When it detects a change on the “activate?” flag, it becomes active, and likewise, a change on the “deactivate?” flag turns it off. When the mission starts up, it goes into the state that you specified in the config zone’s ‘active’ attribute (defaults to true).

WARNING:
WHILE TURNING **OFF** GUARDIAN ANGEL TAKES EFFECT **IMMEDIATELY**, TURNING IT ON (PROTECTING PLANES) ONLY AFFECTS MISSILES THAT ARE SUFFICIENTLY DISTANT FROM PROTECTED AIRCRAFT FOR GUARDIAN ANGEL TO INTERCEPT THEM!

5.4.6.2 Dependencies
Guardian Angel requires dcsCommon and cfxZones

5.4.6.3 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
msgTime	Number of seconds that a warning from Guardian Angel stays on-screen. Defaults to 30 seconds
private	If set to true, all announcements are only made to the group that a missile was fired at. Defaults to false (everyone can see)
launchSound	Name of the sound file to be played when a missile is launched. Respects ‘private’ attribute
interventionSound	Name of the sound file to be played when GuardianAngel saves an aircraft. Respects the ‘private’ attribute.
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) **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

Name	Description
	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
active	The state that Guardian Angel starts up in. True means that it is active, false that it is turned off. Defaults to 'false'
activate? on?	Watchflag to turn on (activate) guardian angel. Defaults to <none>
deactivate? off?	Watchflag to turn off (deactivate) guardian angel. Defaults to <none>

5.4.6.4 ME Attributes

You can selectively add/remove aircraft placed with ME to be included into Guardian Angels watch list by placing a zone over them, add adding the "guardian" attribute:

Name	Description
guardian	**MANDATORY** Tells Guardian Angel how to treat aircraft inside the trigger zone. How it treats them is determined by the value you give to this attribute: • *true* All aircraft inside this zone are protected by Guardian Angel • *false* All aircraft inside this zone will not receive protection from Guardian Angel. Defaults to 'true' (all aircraft inside are protected)

5.4.6.5 Using the module

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

Optionally, add a config zone with ME

5.4.7 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

5.4.7.1 Description

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

5.4.7.2 Dependencies

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

5.4.7.3 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.

5.4.7.4 ME Attributes

None.

5.4.7.5 Using the module

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

5.4.8 Civ Air

5.4.8.1 Description

Civ Air is a high-performance drop-in module to quickly add civilian (neutral) AI flights to a mission. You control which airfields have civilian traffic, how many planes ferry between the airfields, and which planes to use as civilian aircraft (note that DCS currently only has a few truly civilian planes) and which liveries to use. 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 airfields on the map, and these planes will randomly use their available liveries. Map and available airfields are detected automatically.

Additionally, you can add 'international' (off-map) entry- and exit points for civilian flights to have flight appear and disappear at the edges.



HOW IT WORKS

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

To generate a flight, Civ Air goes through several steps: It

- picks an aircraft *type* (e.g., Yak-40) from the list of available aircraft types, and assigns the flight to the neutral faction.
- picks a livery from a list of available liveries for that aircraft type
- picks a starting airfield or "inbound location" (locations – marked with trigger zones - on the map where civilian aircraft "poof" into existence. Usually, you would place them at the edge of the map. This is to simulate civilian traffic that originates from a far-away, off-map location)
- picks a destination airfield or "outbound location" (like inbound locations, except aircraft disappear when they reach the zone to simulate an off-map destination)

On mission start-up, the script also generates active traffic as follows:

- half the maximum number of planes start in the air somewhere between their departure and destination airport.

- the remaining civilian planes start spawning on the tarmac and perform a cold start if they start from an airfield, or start in the air if they “jump in” from an inbound location. One civilian flight spawns per cycle (once every 25 seconds unless changed with a config zone) until the maximum number of civilian flights is filled.

DEFAULT 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 also increase the likelihood of a plane appearing by adding its type multiple times, increasing its chance of being picked over the others.

You control which aircraft types are used with attributes in civAir’s config zone.

LIVERY SUPPORT
CivAir supports “liveries” (paint schemes for aircraft). By default, the module supports the built-in liveries for following aircraft types:

- **Yak-40**
  Aeroflot, Algeria GLAM, Olympic Airways, Ukranian, Georgian Airlines

- **C-130**
  Air Algerie L-382 White, Algerian AF Green, Algerian AF H30 White, Belgian Air Force, Canada's Air Force, French Air Force, HAF gray, IRIAF 5-8503, IRIAF 5-8518, Israel Defence Force, Royal Air Force, Royal Danish Air Force, Royal Netherlands Air Force, Royal Norwegian Air Force, Spanish Air Force, Turkish Air Force, US Air Force

- **C-17A**
  usaf standard

- **IL-76MD**
  Algerian AF IL-76MD, China Air Force New, China Air Force Old, FSB Aeroflot, MVD Aeroflot, RF Air Force, Ukrainian AF, Ukrainian AF Aeroflot

- **An-30M**
  15th Transport AB, China CAAC, RF Air Force

- **An-26B**
  Abkhazian AF, Aeroflot, China PLAAF, Georgian AF, RF Air Force, RF Navy, Ukraine AF

By default, when choosing an aircraft, CivAir also randomly chooses a livery from the list of appropriate liveries. You can supply your own list of aircraft types and liveries to replace the default set of aircraft and liveries.

You can add support for more/other liveries through the a “civil_liveries” zone.

OPTIONAL: SUPPOR FOR “CAM”

There is a popular DCS mod called “Civilian Aircraft Mod” which adds some common civilian aircraft types (e.g.727, A-320 etc.) to your DCS installation, and comes stocked with a host of liveries. DML knows about the types and liveries, and provides instantaneous access to them. Note that DML does neither check nor care if the DCS installation that runs a mission which uses CAM planes really has access to them. Whenever your mission accesses an aircraft type that does not exist (for whatever reasons), there will simply be one civilian plane less in the sky (or, depending on your DCS version, a default civilian Su-27 with a fantasy green livery called ‘placeholder’); there will be no error, the mission chugs on without a hitch. This means that using **civAir does not, in any way, depend on CAM,** and can easily take advantage of it.

If you, as mission designer choose to support CAM in your misisons, it’s up to you to tell your players that they should install CAM, lest they merely see more blue skies instead of some nice civil aircraft.

CONTROLLING TYPES AND LIVERIES

While civAir is designed to function out-of-the-box with a minimum of effort by providing sensible defaults, it also provides a comprehensive and easy-to-use mechanism to tightly control which aircraft types are used with which livery. Key to this are the “civAirConfig” and “civil_liveries” zones:



You start by setting up a trigger zone (anywhere on the map) that must be named “civil_liveries” (exact match require) that lists which aircraft type uses which livery. As QoL, any aircraft type that you list here is also automatically added to the list of available aircraft for civ air (because I assume that if you list a type and its liveries you also want to use them).

NOTE:
So how and where do I get the names of liveries that are available for a type (e.g. YAK-40)? That, unfortunately, is not universally explained, nor is there an easy reference available. To know a livery’s name, you must have the aircraft and livery installed on your computer, and then trawl your DCS installation. Find the aircraft’s folder in your DCS installation’s Bazar/Liveries/ folder (e.g. Bazar/Liveries/Yak-40/). Inside that folder there will be more folders, and each folder’s name found there has a good chance to be the name of a livery.

Any livery that you list for a type will be made available to the listed aircraft.
Note: the list of liveries that you give here **replaces** an existing list of liveries for that type. Use this to curtail/restrict a list of liveries for one of the default aircraft types (DCS and CAM) with your own

At this point, civAir will use the default “DCS” aircraft types and liveries, *plus* any type type that you have listed in the civil_liveries attributes.

You can further broaden/restrict this with the civAirConfig zones attributes as follows:



- Set the attribute “DCS” to false: this will remove the default set of DCS cvil aircraft – unless you explicitly name a type in your civil_liveries attribute (it then remains available to civAir)
- Set the attribute “CAM” to true: this will also add the entire set of CAM aircraft types to the available aircraft types.
- Provide “aircraftTypes” with a list of aircraft types: this completely replaces the list of available aircraft types with your choice (and repetition) of aircraft types. This provides full backward-compatibility, and it allows you to control the probability for any aircraft type to be chosen by civAir when it creates a flight.

DEPARTURE/DESTINATION AIRFIELDS IN CIVAIR

Flights can originate from an airfield or an inbound location, and end at another airfield or outbound location. You can control which airfields can spawn civilian flights.

Airfields are kept in two separate lists: departure points and arrival points. Airports can be both departure and arrival points, and you can elect to have airfields that only serve as departure of arrival points. Airfields that are on neither list are closed to civilian traffic.

You add an airfield in ME as departure or arrival point by adding a ‘civAir’ attribute to a trigger zone. The nearest operational airfield will then be configured as arrival/departure point. Note that if you have two or more trigger zones with the civAir attribute closest to the same airfield, this **airfield will be added multiple times, making it proportionally more likely to be picked** as departure or destination airfield. The value that you supply with the civAir determines how the airfield is treated:

- By default, the airfield is a departure and arrival point. Civilian aircraft depart from here and land here. If you leave the value empty the airfield is open for both arrivals and departures
- A value of “exclude” or “closed” means that this airfield is closed to civilian traffic. This overrides any departure or arrival settings.

DEPARTURE-ONLY/ARRIVAL-ONLY AIRFIELDS

Airfields can be designated as departure-only or arrival-only. To do so,



- supply a value “depart”, “departure” or “depart only”. The airfield is can only spawn departing flights.
- A value of “arrive”, “arrival” or “arrive only” sets the airfield up as destination-only, flights cannot originate here.

INBOUND AND OUTBOUND (OFF-MAP) FLIGHTS

You can also use a trigger zone to denote entry and exit points for flights that enter or leave the map. To do that, place a trigger zone where the flight should disappear from the map, add a ‘civAir’ attribute, and enter one of the following values

- A value of ‘inbound’ or ‘in’ denotes this zone as an entry point for simulated off-map flights. Flights will spawn inside this zone. This zone will not be associated with any airfields. Although the zone can be placed anywhere on the map, it is recommended that you place it close to the map’s border.
- A value of ‘outbound’ or ‘out’ denotes this zone as an exit point for simulated off-map flights. Flights will disappear when they reach this zone. Again, I recommend that you place this zone close to the map’s border.
- A value of ‘in/outbound’ or ‘in/out’ allows flights to both appear and disappear in this zone.

PRO TIP
Flights are named by their departure and destination. For airfields, it’s the name of the airfield. For in- or outbound locations, the trigger zone’s name is used. This means that if you give them some meaningful names (e.g. “New York” for a far west in/outbound zone on the Sinai map), the flight looks much more meaningful and interesting.

AVAILABILITY OF AIRFIELDS
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.

DEFAULT BEHAVIOR
If you do not add any airfields by placing a trigger zone with a ‘civAir’ attribute (this includes in/outbound zones), civAir defaults to adding *all* airfields to the map to instantly populate the entire map with civilian traffic. If you add at least one airfield where flights can depart from, and at least one airfield where flights can land, then only those airfields are used for civil flights. This allows you to quickly define all airfields that are serviced by civAir flights (i.e. adding two civAir zones to your map is sufficient to control all flights).

IMPORTANT
CivAir requires at least one airfield/zone to depart from, and one airfield/zone to fly to. If you add too few departure or arrival points, CivAir adds all airfields on the map as both destination and arrival. If, for example you only add a single off-map inbound zone, CivAir will add all airfields to that, and civilian traffic can depart from any airfield and the inbound zone, and all airfields are a valid destination.

If, on the other hand, you add a single inbound zone and a single arrival only airfield, all traffic will be from the inbound zone to the arrival-only airfield.

FLIGHTS
Flights either start cold at ramp, or hot in the air (from an inbound zone, or at mission start). After cold-start, aircraft taxi and depart the airfield. Flights proceed to their destination. When heading to an outbound zone, they simply disappear when they are within 5km of the outbound zone’s center.

When heading to an airfield, the first overfly the destination airfield, and then land there. Once landed, they taxi to their parking lot, and power down. After some delay, they de-spawn, and a new flight can be created.

To populate a mission's airspace when the mission begins, you can allow aircraft to appear some place between their origin and destination.



Flights (their group) are named dynamically by their origin and destination. When the origin or destination is an airfield, the airfield's name is used. When flights start or end in inbound or outbound zones, the zone's name is used instead. In the example above, we see a flight that entered the map from the 'in/outbound' trigger zone named 'Zürich' and is enroute to Sochi-Adler.

AUTO-DIVERT
If a flight can't land on the intended airfield (for example, if a C-17 is scheduled to land at an airfield that can't handle an aircraft of that size, or because the runway is destroyed), the aircraft will automatically divert to the nearest airfield that can.

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.

5.4.8.2 Dependencies
CivAir requires dcsCommon and cfxZones

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

- Place a Trigger Zone anywhere in ME
- Name it "civAirConfig" (note: name must match exactly)
- Optionally, add a zone named "civil_liverioes" and add attributes that list aircraft types and liveries
- Add any of the following attributes to this zone:

Name	Description
verbose	A value of "true" turns on debugging messages. Default is "false"
DCS	By default, civAir adds a set of standard DCS aircraft types (Yak-40, C-130, C-17A, IL-76MD, An-30M, An-26B) to its list of civil aircraft. You can turn that off by setting the attribute's value to false

Name	Description
	Defaults to 'true' (add above mentioned standard DCS types)
CAM	If you set this attribute to true, civAir also adds the following airfcaft types to its list of available aircraft: A_320, A_330, A_380, B_727, B_737, B_747, B_757, Cessna_210N, DC_10 These aircraft types aren't included in a standard DCS install, and using such an aircraft type simply results in a non-spawning civil aircraft. The types listed above are aircraft that come with a popular mod called "CAM". If that mod isn't installed when DCS runs a mission that tries to access such an aircraft type, the relevant flight simply will not spawn. Defaults to false (no CAM types added tp civAir)
aircraftTypes	A comma-separated list of Types (as defined in https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB/Aircraft) that define the aircraft types used for civilian flights. These must be fixed wing aircraft (i.e not helicopters). **When present, 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. If, for example, you list `Yak-40, Yak-40, IL-76MD` During the mission two thirds of all civilian flights will be performed by Yak-40, and the remaining third by IL-76MB. The list that you provide here overrides any list that was assembled by the attribute values for "DCS" and "CAM" (see above), or provided in a civil_liveries zone. If you accidentally misspell a type name, or that type does not exist for your DCS installation, DCS will to either of two things: • Create no flight at all (some versions of DCS) • Create a civilian flight of a SU-27 with fantasy "placeholder" livery – or some other random airframe/livery combo)other versions of DCS) Neither will disrupt your mission in any way, so using this feature is safe.' Defaults to <none> (aircraft used are taken according to the settings of the attributes DCS and CAM, and whatever you supply in the civil_liveries zone
owner	Country ID that all aircraft spawned by civAir belong to. Defaults to 82 (UN Peacekeepers)
ups	Number of updates per second that civAir checks on its flights. By default, this is 0.05, or once every 20 seconds.
maxTraffic maxFlights	Maximum number of civilian flights at the same time. Defaults to 10

Name	Description
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. Defaults to TRUE (aircrafts can spawn mid-air) at mission start.

civil_liveries
With this zone you can add any aircraft type and a list of liveries to be used for that type

Name	Description
Type_name Example: YAK-40	List if livery names, separated by comma. Example: Aeroflot, Algeria GLAM, Olympic Airways

You can add as many type names as you like, and as many liveries that you like. If a type does not exist, that flight will not spawn. If type *does* exist, but its livery doesn't, civAir falls back to DCS's default livery for that aircraft type. The flight will spawn, just not in the desired livery.



5.4.8.4 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 can be added to civ air's airport list. The following values are supported: • 'closed' or 'exclude' No civilian flights originate or land here. This overrides any other settings for this airfield • 'departure' or 'depart only' Civilian flights can depart only from this airfield • 'arrival' or 'arrive only' Civilian flights can only land here • 'in' or 'inbound' – DOES NOT ASSOCIATE AN AIRFIELD Civilian flights can appear in this trigger zone, in-flight and at altitude. This is intended to be used to simulate off-map long-distance flights that now enter the map arena. Instead of an associated airfield's name, this zone's name is used for the

Name	Description
	flight's origin • 'out' or 'outbound' – DOES NOT ASSOCIATE AN AIRFIELD Civilian flights that reach this trigger zone disappear. This is intended to simulate an off-map connection, where a long-distance flight leaves the map. Instead of an associated airfield's name, this zone's name is used for the flight's origin • 'in/out' or 'in/outbound' – DOES NOT ASSOCIATE AN AIRFIELD Civilian flights can both appear in this trigger zone, and disappear when they reach it as their destination. This is intended as both a source and destination point for flights that have an off-map source and/or destination. Instead of an associated airfield's name, this zone's name is used for the flight's origin or destination • *Any other value, or nothing* The airfield closest to this zone is an open airfield for departing and arriving civilian flights Note that you can add the same airfield multiple times by adding multiple trigger zones with a civAir attribute in proximity the same airfield. This will proportionally increase the likelihood that the airfield is picked for destination or departure Note also that only functioning airfields are chosen. FARPS or ships are disregarded. **MANDATORY**

5.4.8.5 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.

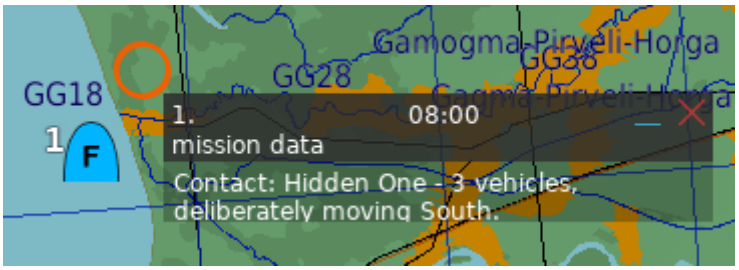
You can then customize that setup in the above-described way.

5.4.9 Recon Mode

5.4.9.1 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’ – groups that have been assigned a priority for the recon planes to find
- supports a ‘blacklist’ – groups that recon planes will never find (the can’t see them even if standing next to them)
- provides special messages when priority units are detected
- 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)
- report detected units to the scout’s coalition
- ability to mark detected units on the F10 map
- can be turned on and off via input (flag signals)



Reporting In

Whenever a scout detects an enemy unit, this is reported to all players of the scout’s side. This feature can be turned off. The report is made in the following format:

- Strength: number of units (infantry / vehicles)
- Action: if they are stationary or in which direction they are moving
- Location: in Lat/Lon (default) or MGRS
- (when twn is loaded in mission) distance and direction to nearest town
- Time: when they were observed



Additionally, a group that is marked as priority (see later) can have an individual message that is displayed upon the group’s discovery. The message is defined in the priority-designating trigger zone, and supports most of the same formatting as the ‘messenger’ module

Note that you can selectively suppress scout reports for groups by using the ‘silent’ attribute for priority groups (see later)

F10 Map Marks
Whenever Recon Mode detects enemy units, it places a mark on the F10 map that all players on the same side can see. The mark optionally contains additional information (how many units sighted, group name, action). The mark does not update, and therefore represents the initial contact location and strength. The mark 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 an attribute in the config zone

Notes:

- placing a mark can be suppressed for groups by using a 'silent' attribute for priority groups.
- Marks can be removed by players unless you set the config zone's "marksLocked" attribute to true, in which case marks remain on the map until they are automatically removed.

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.

ME INTEGRATION
Like most other stand-alone modules, ReconMode uses a config zone to curtail its base behavior. Additionally, it can use zones to curtail which aircraft can perform recon, and which ground forces can be detected:

Adding Recon Planes to a mission: "Scout Zones"
Using trigger zones placed over aircraft in the Mission Editor, you can designate planes as scouts (i.e., they actively report any new enemy ground forces that they detect), or 'no-scouts', meaning that they lack the ability to perform recon during the mission.

This is important because for ease of use, ReconMode can be set up automatically allow or disallow all aircraft to perform recon, and you would use the scout option to add planes as scouts (when disabling all planes except special scouts), and the no-scout option to remove recon ability from some planes (when the default is to allow scouting for all planes)

To assign scout or no-scout abilities to units, place aircraft (not ground units) inside a trigger zone with the 'scout' attribute:

Name	Description
scout	Marks all aircraft (fixed- and rotor-wing) inside the zone. If the attribute's value is 'true', the aircraft have recon ability. If the value is false, they are 'blind', i.e. they have no recon abilities. Defaults to 'true' **MANDATORY**
dynamic	Controls if all units that start with the same name are automatically included. This is helpful for clone zones that base all names for clones on the name of the unit in the template.

Name	Description
	Defaults to false

Further modifiers affecting scout aircraft: AutoRecon
In addition to “Scout Zones”, basic recon behavior of all planes is governed by the module’s config zone (see later). Particularly of interest here is the autoRecon mode, which is enabled by default. In that case, there are a couple of easy modifiers that curtail recon ability

- **redScouts / blueScouts / greyScouts**
  Selectively enables/disables autoRecon for that side

- **playerOnlyRecon**
  Grants recon abilities only to player (i.e., not AI) aircraft. Note that playerOnlyRecon is applied *after* red/blue/grey modifiers

Modifying Ground Troop Visibility (“Recon Zones”)
Recon mode usually reports all groups of enemy units that it detects. You can modify this behavior by changing how some groups are reported:

- **Priority Groups**
  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 banged. This allows you to designate special groups that can elicit different responses from your mission depending upon which group was discovered.

- **Flag events when groups are spotted**
  ReconMode provides multiple ways to alert your mission that groups have been spotted by a recon aircraft:
  - via the global detect! flag that is triggered whenever a group of ground forces is detected that is not a priority (set with the config zone)
  - through the global prio! flag that is triggered whenever a group of ground units is detected that is marked as priority
  - through the optional spotted! flag that can be set individually for each priority target

- **Invisible (blacklisted) Groups**
  ReconMode 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 must be discovered the old-fashioned way.

Priority Messages
When you designate groups as priority groups, you can optionally trigger a ‘priority message’ that Is displayed. This message is individual per trigger zone, and supports the following formatting wildcards:

- <n> creates a new line
- <z> is replaced with zone’s name
- <t> is current time in HH:MM:SS format
- <lat> the latitude of the discovered group’s current position

- <lon> the longitude of the discovered group's current position
- <ele> the elevation (in feet or meters, as determined by the imperialUnits reconModeConfig attribute)
- <mgrs> the discovered group's position in MGRS coordinates

Only applicable for priority targets (i.e. recon's value is something other than "black"), ignored otherwise.

Note that the reported location is that of the first surviving unit found in the group. If your group is spread over a significant area, the returned position may be misleading (but it's still the location of one of the units in that group).

Note also that priority messages (when defined) will be displayed even when a 'silent' attribute is present (see below). That way you can output a message without giving any other indication why or where the message was generated.

(Blacklisted groups do not have that feature since they by definition aren't spotted by recon aircraft)

Staying Silent (no report, no mark)
You can suppress both recon report and map marks for any groups inside a priority zone by adding a <silent : true> attribute to a priority zone (does not work with blacklisted groups, as they never generate a report nor map mark).

Suppressing the report and mark can be desirable if you wish to use recon mode simply to trigger an event in your mission and still want to use normal reporting (i.e., keep the config's 'announcer attribute set to true).

Support for Cloners
ReconMode has built-in support for cloners: when you designate ground forces as priority targets or blacklist them (making them invisible to scouts), a simple option allows you to treat all dynamically spawned clones of that group the same way.

Adding Ground Units/Groups to Priority- or Black-List
Recon mode supports adding groups to black-or priority lists with trigger zones. Simply add the "recon" attribute with either "black" or "prio" as value, and all groups inside that zone will be added to the relevant priority list.

When you mark a zone as a priority target, you can use additional attributes to customize the message that is sent or flags that are banged

Name	Description
recon	Marks all ground groups inside this zone as recon relevant. If the value is "black" all groups that have at least one inside this zone are added to the blacklist, otherwise they are added to the priority target list Defaults to "prio" **MANDATORY**
dynamic	Controls if all groups that start with the same name are automatically included. This is helpful for clone zones that base all names for clones

Name	Description
	on the name of the unit in the template. Defaults to true
prioMessage	A message that is displayed to the coalition that the scout belongs to. Can contain most of the text wildcards that the messenger module provides: • <n> creates a new line • <z> is replaced with zone's name • <t> is current time in HH:MM:SS format • <lat> the latitude of the discovered group's current position • <lon> the longitude of the discovered group's current position • <ele> the elevation (in feet or meters, as determined by the imperialUnits attribute in reconModeConfig) • <mgrs> the discovered group's current position in MGRS coordinates *Only applicable to priority targets* (i.e., recon's value is something other than "black"), ignored otherwise. Default is <none>, i.e.., no message is displayed
spotted!	DML output flags to bang when a group is spotted. Flag is banged *in addition* to the module's global "prio!" flag. *Only applies to prio zones* (ignored when blacklist zone) Defaults to <none>
silent	Only applicable to priority groups. When present and set to true, the recon report and map mark are suppressed for any groups defined with this zone. Note that this does NOT apply to any prioMessage you have defined for this zone, as that will still be displayed. Defaults to false

Turning Recon Mode on and off during the mission
Recon Mode can listen to flags for a change to turn its scouting abilities on and off. You define these flags in the config zone.

Performance Considerations
To do its job, a recon function must regularly check all existing troops on the ground against all scouts: 100 units on the ground and 4 scouts means that 100 * 4 = 400 checks are required regularly. This can quickly escalate in terms of performance requirements. Recon Mode uses several methods to intelligently limit its performance drain on the mission:

- Recon planes perform their checks only every few seconds, not permanently
- Checks are spread over time, not all at once
- Neutral troops are ignored
- Visibility checks are optimized (e.g., if one unit of a group is too far away to detect it is assumed that they all are since groups move in proximity to each other)

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 limitations 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.

5.4.9.2 Dependencies

Recon Mode requires dcsCommon and cfxZones

5.4.9.3 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 (all planes behave as scouts and report enemy units)
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 (all blue air units are scouts)**
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 NOTE:** 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
marksFadeAfter	Time (in seconds) after which a mark on the F10 map is

Name	Description
	automatically removed. Set to a negative value (e.g., -1) for 'eternity' (marks never time out) Defaults to 30*60 = 1800 seconds (= 30 minutes)
marksLocked	When set to true, marks cannot be removed by players, and will disappear automatically after they time out (see marksFadeAfter, above) or the group is destroyed (when autoRemove is true) Defaults to false (player can remove marks)
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!	DML flag to bang! when a priority unit is detected Defaults to <none>
detect!	DML flag to bang! when an enemy group is detected. If that same group is on the priority list, prio! is banged instead. Defaults to <none>
method reconMethod	DML method for output flags. Defaults to 'inc'
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
mgrs	Defines if the location of the group that is detected is given in Lat/Lon (default) or MGRS. Set to true to enable MGRS. Default is false (coordinates are displayed in Lat/Lon)
imperial imperialUnits	When set to true, the value given in <ele> (elevation) is in feet, otherwise in meters Defaults to false (*ele* is returned in meters)
activate? on?	Watchflag to monitor for a change. If a change is detected, Recon Mode goes into active state. Defaults to <none>
deactivate? off?	Watchflag to monitor for a change. If a change is detected, Recon Mode turns off Defaults to <none>
active	Set to false to start Recon Mode in disabled (off) state. Defaults to true (Recon Mode enabled)
reportTime	Number of seconds that a recon message stays on the screen. Defaults to 30.
groupNames	Set to false if you want a SALT report (and the ground marks) to omit group names. Defaults to true (group names are reported)

5.4.9.4 ME Attributes

To designate groups of ground forces as either priority targets (which allows for special flag signals) or invisible to recon aircraft, place them inside a trigger zone and use the following attributes:

Name	Description
recon	Marks all ground groups inside this zone as recon relevant. If the value is "**black**" all groups that have at least one unit inside this zone are added to the blacklist, otherwise they are added to the priority target list Defaults to "**prio**" **MANDATORY**
dynamic	Controls if all groups that start with the same name are automatically included. This is helpful for clone zones that base all names for clones on the name of the unit in the template. Defaults to true
prioMessage	A message that is displayed to the coalition that the scout belongs to. Can contain most of the text wildcards that the messenger module provides: • <n> creates a new line • <z> is replaced with zone's name • <t> is current time in HH:MM:SS format • <lat> the latitude of the zone's current position • <lon> the longitude of the zone's current position • <mgrs> the zone's current position in MGRS coordinates *Only applicable for priority targets* (i.e. recon's value is something other than "black"), ignored otherwise Default is <none>, i.e.., no message is displayed
spotted!	DML output flags to bang when a group is spotted. Flag is banged *in addition* to the module's global "prio!" flag. *Only applies to prio zones* (ignored when blacklist zone) Defaults to <none>
silent	Only applicable to priority groups. When present and set to true, the recon report and map mark are suppressed for any groups defined with this zone. Note that this does NOT apply to any prioMessage you have defined for this zone, as that will still be displayed. Defaults to false

To assign or remove scouting abilities to aircraft, place them inside a trigger zone with the following attributes:

Name	Description
scout	Marks all aircraft (fixed- and rotor-wing) inside the zone as potential scouts. If the attribute's value is 'true', the aircraft have recon ability. If the value is false, they are 'blind', i.e. they have no recon abilities. Defaults to 'true' **MANDATORY**
dynamic	Controls if all units that start with the same name are automatically included. This is helpful for clone zones that base all names for clones on the name of the unit in the template. Defaults to false

Note that this includes player planes.

5.4.9.5 Using the module

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

Optionally, add a config zone with ME

5.4.10 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 SSB's slot-blocking ability in their missions to block slots by faction. This means that by including ssbClient into your mission you can use intelligent slot-blocking in your missions.

Note that **ssbClient works best with SSB but does *not require* SSB**. Since SSB is a server module, it needs the mission to be run in multiplayer mode on the SSB-enabled server. If you add the '**slotty'** module to your mission, you can ensure that slot-blocking also works for single player *and* when SSB is not available on the server.

NOTE
The '*slotty*' module allows single-player slot-blocking, and provides a fallback for multiplayer slot-blocking if SSB is not installed on the server. So always add 'slotty' to your missions when you add ssbClient to provide smart slot blocking for single-player and servers that host your mission but did not install SSB

5.4.10.1 Description

ssbClient provides **automatic slot blocking** for aircraft that

- have their starting location on an airfield/FARP that currently belongs to the enemy (optionally neutral as well). This is a *dynamic* feature, and when an airfield is captured, the aircrafts based there change availability (free or blocked)
- are on an airfield that is "closed". ssbClient provides DML watchflags/Zones (and a script API) to open/close airfields
- (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 the module ssbSingleUse provides a better, more flexible support for this.

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. For this reason, I recommend that you only use single-unit groups with player (client) planes.

Automatic Slot-Blocking /-Enabling on Capture
Slot blocking and enabling is fully automatic. ssbClient manages all airfields and FARPs by their owning faction. If an aircraft originates from an airfield/FARP, the slot is

- **blocked** if the airfield does not belong to the same faction
- **enabled** if the airfield belongs to the same faction
- you can **allow neutral** fields to count as allied to both sides (using the allowNeutralFields attribute in the config zone). In that case all aircraft on neutral airfields are enabled. If a neutral airfield is captured by either side, the other side loses access to that airfield

If a slot is blocked, a player can no longer use that slot from the change role dialog.

“Caught in the air”
If a player successfully slots into an aircraft, and the originating airfield subsequently is captured, they can remain in that aircraft (i.e. they won’t be kicked from the aircraft). Once they relinquish that slot (voluntarily, or by crash), it only becomes available again once the airfield is recaptured.

Opening / Closing Airfields
ssbClient provides ME integration for “opening” and “closing” airfields. An airfield that is closed can be landed on, and taken off from, but any aircraft slot that originates from a closed airfield is blocked until the airfield opens, no matter which faction owns the airfield/FARP. You can use the open? and close? attributes to control opening and closing an airfield when you use ssbClient zones.

Note that an open airfield still adheres to ownership rules, so a blue aircraft can’t spawn on an opened red-owned airfield.

Block aircraft after Crashing (“using up” aircraft)
ssbClient provides mission designers with the option to block a slot after that aircraft was destroyed. This allows for game mechanics where aircraft can get used up, so players become more risk-averse in their flying, lest they lose access to that aircraft.

If you intend to use this feature, you must first edit the (server-side) SSB source, and change the following

```
ssb.kickReset = true
```

to

```
ssb.kickReset = false
```

Note that I recommend that you always set ssb.kickReset to false, as it integrates better with DML, and poses no drawbacks.

No SSB? No Problem! – Call ‘slotty’!
ssbClient conforms to the protocol that Ciribob invented with his “SSB” simple slot block server module. The central idea in this module is that a group that should not be made available to players is marked in a special way. ssbClient adheres to this protocol and marks aircraft on airfields accordingly, if the owner of the airfield doesn’t match the one of the aircraft. If SSB is running on the server, that slot is blocked.

But what if SSB isn’t running on the server or the mission is run local/single-player? In that case there is no SSB that can block slots for you. But there is an easy fallback: “Slotty”. I recommend that whenever you add ssbClient to your mission to add smart slot blocking, also add “slotty” to provide single-player slot blocking and a fallback for multiplayer if SSB isn’t running on the server.

5.4.10.2 Dependencies
ssbClient requires dcsCommon, cfxMX and cfxZones

5.4.10.3 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 "SSBClientConfig" (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 (remain blocked)
allowNeutralFields	If set to "true", aircraft can spawn on neutral airfields (otherwise they are blocked). Defaults to false (neutral fields do not allow blue nor red aircraft to spawn
maxAirfieldRange	Maximum range in meters to find an airfield/FARP for a 'from ground' start. If no airfield is found that slot will be permanently open. Defaults to 3000 meters
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
ssbAutoenable	When set to false, ssbClient doesn't signal to SSB by itself that it should activate. Your mission then either has to enable SSB by itself, or use the *slotty* module as slot-blocking mechanism (note that I

Name	Description
	recommend to always add slottty to missions that use ssbClient to provide a fallback when the mission is played single-player or SSB isn't running on the server). Defaults to true – enable SSB for the mission.

5.4.10.4 ME Attributes

You can use Trigger Zones with the ssbClient attribute to control slot access to airfields/FARPs

Name	Description
ssbClient	Marks this zone as a control zone **for the airfield/FARP that is closest** to this zone. **MANDATORY**
open?	DML Input Watchflag. When triggered, this airfield "opens", allowing planes spawn from this airfield. Note that ownership rules still apply. Defaults to <none>
close?	EML input Watchflag. When triggered, this airfield/FARP "closes". Aircraft can still land and depart from a closed airfield, but player aircraft that originate from there will no longer be able to spawn, their slots are blocked. Defaults to <none>
openOnStart	When set to false, the airfield is closed on mission start, all slots for aircraft spawning here are blocked Defaults to true (airfield is open, allowing slots to spawn according to faction ownership of airfield/FARP)
ssbTriggerMethod	Method for Watchflags. Defaults to "Change"

5.4.10.5 Persistence

SSBClient correctly persists which slots are blocked, and which airfields/FARPs are closed, so when you load the mission, they will assume the same status as the point in time when the mission was saved. This includes slots blocked due to loss of aircraft.

5.4.10.6 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.

To open/close airfields/FARPs with flags, use Trigger Zones with the ssbClient attribute

5.4.11 ssbSingleUse (tbc)

5.4.11.1 Description

5.4.11.2 Dependencies

5.4.11.3 Module Configuration

5.4.11.4 ME Attributes

5.4.11.5 Using the module

Remember to turn off kickReset

5.4.12 unGrief (enforce PVP/PVE mode, no friendly kills)

Griefer:
A “griefer” or bad faith player is a player in a multiplayer video game who deliberately irritates and harasses other players within the game (trolling), by using aspects of the game in unintended ways, such as destroying something another player made or built. A griefer derives pleasure primarily, or exclusively, from the act of annoying other users, and as such, is a particular nuisance in online gaming communities.

5.4.12.1 Description
unGrief is a module that aims to discourage friendly kills in multi-player missions, the actions of so-called ‘griefers’, i.e. asocial elements that get a rise out intentionally disrupting the game with friendly fire. When friendly kills are detected, the player responsible for the kill is warned, and if they continue their actions, their plane is destroyed. If they continue to kill their own side, their plane can be immediately destroyed/disabled when they try to occupy a slot during that same mission (everything is forgotten when the mission ends)

unGrief works out-of-the-box.

PLAYER-LEVEL ENFORCEMENT
unGrief records infringing players by keeping their log-in Name (player handle) on a black-list, so it tracks a player even if they switch planes, sides, or log out and then log back on - as long as they do not switch their login name, unGrief recognizes and remembers them. That being said, unGrief does not persist offenders after the mission ends, so when a mission is re-started, earlier misdeeds are forgiven and forgotten.

PVE-RULES (NO PVP ALLOWED)
unGrief allows you to enforce a PvE-only play style. In PvE, all player versus player kills are forbidden and count like killing a friendly. To use unGrief to simply disallow player-player killing, use the setup shown on the right.

Name	Value	
pve	yes	
ignoreAI	yes	

PvP ZONES IN PvE MISSIONS
For servers, it may make sense to set PVE rules, and still allow PVP in designated areas. unGrief has a very simple way of allowing this: simply add a zone with an ‘pvp’ attribute, and PVP is allowed inside that zone. Note that all other unGrief options are still in effect: blue on blue or red-on-red kills are still not allowed. Multiple PvP zones per map are allowed.

STRICT AND RELAXED PVP RULES
To determine if a PVP kill is legal, the plane’s positions **at the moment of the kill** are relevant. Usually, if a player unit is killed by another player while it resides inside a PvP

zone, the kill is legal even if the killing plane has fired from outside the zone. This is called ‘relaxed’ PvP rules, and it allows players to kill planes inside a PvP zone without entering it themselves.

To prevent ‘safe snipers’ loitering just outside a PvP zone and kill players inside, you can opt to turn on ‘strict’ PvP rules per zone, in which case both planes must be inside the PvP zone at the moment of the kill. If either plane is outside, the kill is illegal.

If, a player unit is shot down outside a PvP zone, it’s always an illegal kill - even if the shooting plane is inside a PvP zone. Therefore, be wary when shooting at players that are running from you towards safety, lest they reach safety and you inadvertently score an illegal kill.

IGNORE AI (NON-PLAYER UNIT) KILLS
unGrief can ignore all Player-AI kills. This is useful when you want to use unGrief to enforce PvE rules, but relax AI (non-player unit) killing.

GRACE KILLS
unGrief grants a number of kills per player before it retaliates for the first time. After that limit of ‘free’ friendly kills is used up, each friendly kill will have the player’s plane destroyed or the player kicked (when using SSB).

RETALIATION: BOOM vs SSB
unGrief knows two methods to retaliate for transgressions:

- ‘boom’ – to quote an ancient game: you are set up the bomb. The player’s plane is destroyed by an explosion. This also works in single-player mode (playtesting). Although highly gratifying to watch, this can be harmful to other players nearby.
- SSB – unGrief signals SSB that the player’s plane is to be kicked. This *requires* that the mission is run as multiplayer, *and* SSB is installed on the server. Does not work in singleplayer (play-testers take note)

A WRATHFUL SCRIPT
unGrief can be set to be wrathful towards repeat offenders: the first time that a player is killed by unGrief for a griefing transgression, they may re-slot and continue. If they should transgress again by killing friendly units, unGrief becomes wrathful against them, and destroy their plane as soon as they re-slot. Every time. Until the mission ends. No exceptions. For “*though shalt tremble before the mighty unGrief until the day is done and everything complete!*”

SIDE EFFECTS
Be advised that there are situations and mission set-ups where accidental blue-on-blue or red-on-red kills can happen, especially when using AI crew (Petr, George) or cluster munitions in close vicinity to your own troops. Helicopter pilots also be careful when dropping off or picking up Infantry, for accidentally squishing your own troops can trigger swift retribution from a wrathful unGrief script.

LIMITATIONS
If a transgressing player switches out of the airframe that is used for the kill before the kill registers (a kill can take some time to register when the unit 'cooks off'), they escape the consequences.

5.4.12.2 Dependencies
unGrief requires dcsCommon and cfxZones.

5.4.12.3 Module Configuration
unGrief uses a configuration zone for setting up all options. To configure this module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "unGriefConfig" (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"
graceKills	Number of own faction ("friendly") kills that are permissible before unGrief retaliates. Set to 0 to disallow (and immediately punish) any friendly kills. Default is 1
retaliation	How unGrief retaliates towards the player when their graceKills are exceeded. The following options are supported: • 'boom' Place a small explosive inside the plane, grin, ignite. • 'ssb' Use server-side SSB to kick the player from the plane Default is 'boom'
wrathful	Sometimes griefers are slow learners. When set to true, unGrief disallows a repeat offender to re-slot after their second transgression. Their plane is destroyed every time they try to slot until the mission ends. Defaults to false
pve pveOnly	When set to true, PVE rules are in force for this mission. Player versus Player kills outside of (optional) PVP zones count like friendly kills. Defaults to FALSE (PVP killing is allowed across the map)
ignoreAI	Ignores friendly kills if the killed unit was AI-controlled (i.e. not a player unit). Useful in conjunction with PVE rules. Defaults to false.
warnings	When set to true, players entering and leaving a PvP zone receive a notification. Only works when PvE is set to true Defaults to true

5.4.12.4 ME Attributes
unGrief uses config zones for all settings. In PVE mode, unGrief can use PVP zones as follows:

Name	Description
pvp	The mere presence of this attribute marks this zone as a PVP combat zone. Killing players of another faction in this zone is legal. Same-faction kills are still illegal and will be punished.
strict	When set to true enforces 'strict' pvp rules, requiring both planes be inside the PvP zone **at the time of the kill**. When false, only the killed player plane must be inside the PvP zone. Defaults to false

5.4.12.5 Using the module
Include the unGrief source into a DOSCRIPT Action at the start of the mission

Optionally, add a config zone to set module-global options.

When using SSB as retaliation for griefing behavior, remember that this only works in MP mode, and requires the SSB script be installed on the server.

To allow PVP combat on a PvE server, designate PvP zones with trigger zones, and add a pvp attribute to those zones.

5.4.13 Willie Pete (FAC - Artillery for Targets marked with Smoke)

5.4.13.1 Description

This module provides a complete, multi-player capable solution to simulate FAC operations for aircraft: players mark targets with WP (smoke) rockets, and then tell the artillery to fire on those targets. The module implements all logic, player communication and fire control. If your mission also uses PlayerScore, integration with that module is automatic.



Nerd Stuff:

"Willie Pete" is the slang term for the White Phosphorous munitions that is used to create the highly visible smoke that marks the target; it derives from the WWII phonetic alphabet's designations for "W" (William) and "P" (Peter).

PLAY FLOW

In order to tell artillery where to fire to, a player goes through the following steps:

1. *CHECK IN*
   When the player is less than some 10km outside of a WP target zone, they check in via communications. The plane is now observed by the artillery command.
   Roger Frogger WP One, WP Proving Grounds Target Zone tracks you, standing by for target data.

2. *MARK TARGET*
   The player then proceeds to mark the target(s) inside the WP Zone with a smoke marker (usually using rockets).

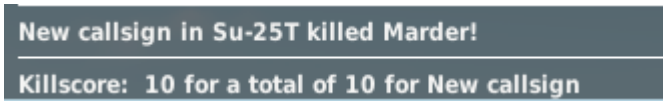
3. *TRANSMIT TARGET DATA*
   When the target is marked with smoke, the player tells Artillery to fire on the smoke marker.
   WP Proving Grounds Target Zone, Frogger WP One is transmitting target location. Fire at 37T GG 32307 36050, elevation 10 meters, target marked.

If the Artillery is ready to fire, they'll fire a salvo of shells onto the location transmitted

4. *IT'S RAINING DEATH*
   When the shells land a short while later, they can destroy the units they with their blast.



5. *INTEGRATION WITH PLAYERSCORE*
   If the mission also uses the PlayerScore module, killed vehicles by the artillery barrage are awarded to the player who placed the smoke.



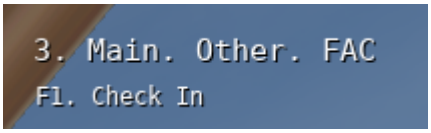
LAST MARKER COUNTS
Whenever a player tells the Artillery to fire on a marker they fired, the last marker they successfully landed in the WP Zone is used. If, for example, the player fires a salvo of multiple WP rockets, the last one that landed inside the WP Zone will become the artillery's target. Note that if there are multiple players in a WP zone, *each player's* last mark is remembered *individually*.

MARKER TIME OUT
WP markers only last for a few minutes (you can change this time with an attribute in the module's config zone) before they become 'stale' and can't be used any more to mark targets. While not timed out, players can instruct the artillery to fire again at the marker.

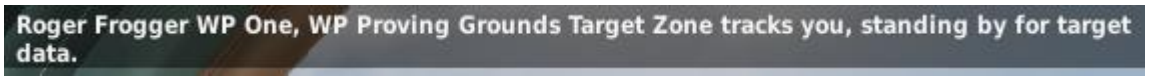
ARTILLERY COOLDOWN / RELOAD TIME
After the artillery has fired a salvo, they require a cool-down time to simulate a re-load. This cool-down can be set per WP Zone individually to simulate different types of artillery, or create a better game experience. Each WP Zone has their own dedicated (simulated) artillery standing by

ALL PLAYERS IN ZONE
If there are multiple players in the same WP and marking targets, any player with a valid (not timed out) marker can command the artillery to fire. The artillery then fires on that player's marker, and all other players must wait for the cool-down to time out before they can command the artillery to fire at their marker.

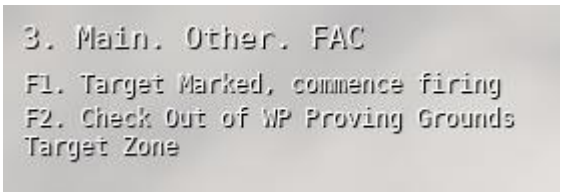
SIGNING IN TO A WP ZONE
In order for a player to successfully mark a target, they must be signed into that zone, and the WP munition must land inside the trigger zone.



To give players enough time to fence in, they have a 'check-in range' that is added to the zone's radius, and defaults to 10km. When inside that range, players can sign into the zone using communications→FAC→Sign In.
If there are multiple zones in range, the player signs into the closest WP Zone. When signed in successfully, the player receives a message that confirms the name of the zone they signed in to



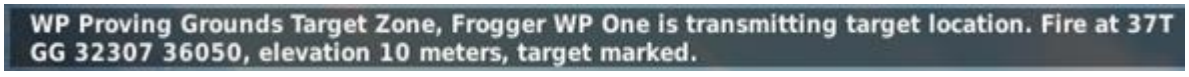
Once signed in, players can call artillery to fire on a marker that they placed within the zone, or sign out of the WP zone.



If players wish to sign into a different WP Zone, they must either sign out first, or leave the check-in range radius (where they are checked out automatically), and close to check-in distance for the other WP Zone before checking in.

Note that the Sign Out command always show the name of the zone that you are currently signed into, so if a player is unsure which zone they are currently checked into, they can simply open the FAC menu and check.

CALLING THE SHOTS
Once a player is signed into a Zone and has placed a target marker (a WP munition), they use the communication→Other→FAC menu and choose the "Target Marked…" command. This places a message on the screen with the target co-ordinates in MGRS.



This message can be seen by every player in the same coalition. A few seconds later, a confirmation message appears:



WP MUNITIONS
In order to mark targets, aircraft must use WP munitions that should land close to their intended targets. Currently the following munitions are supported:

- HYDRA_70_M274
- HYDRA_70_MK61
- HYDRA_70_MK1
- HYDRA_70_WTU1B
- HYDRA_70_M156
- HYDRA_70_M158
- BDU_45B
- BDU_33
- BDU_45
- BDU_45LGB
- BDU_50HD
- BDU_50LD
- BDU_50LGB
- C_8CM
- SNEB_TYPE254_H1_GREEN
- SNEB_TYPE254_H1_RED
- SNEB_TYPE254_H1_YELLOW
- FFAR M156 WP
- HYDRA_70_M259

Any other munitions are ignored.

WHICH AIRCRAFT CAN FAC?
By default, any aircraft that can carry one of the WP munitions listed above can be used to mark targets. You may limit which airframes are accepted as FAC by using the 'facTypes' attribute in the module's configuration zone (see below).

LIMITATIONS

- Currently, WP zones must be circular. Polygonal zones aren't yet supported.
- If you place multiple WP Zones close to each other, be sure to tightly control their individual check-in range to avoid player confusion when they sign in, They may unwittingly sign into the wrong zone.
- WP does not fully support multi-unit player groups. Make sure that each player has their own group. Once DCS supports player-individual menus, this limitation can be lifted.

5.4.13.2 Dependencies
WilliePete requires dcsCommon, cfxZones and cfxMX.

Optional: PlayerScore (no load time limitation)

5.4.13.3 Module Configuration
WilliePete can use a configuration zone to set up certain behavior. To configure it via a configuration zone,

- Place a Trigger Zone in ME anywhere on the map
- Name it “wpConfig” (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”
facTypes	Controls which aircraft can be used for marking targets inside a WP Zone. You can list multiple aircraft types, separated by comma. This attribute also supports the following special types: • ALL or ANY (default, all aircraft are allowed) • HELO or HELI or HELICOPTERS (all rotor-wings) • PLANE or PLANES (all fixed-wing) **Example:** “HELO, A-10A” would allow all helicopters who can load WP ammo, and any A-10A to sign into a WP Zone. Defaults to “ALL”
wpMaxTime	Time in seconds for how long a smoke marker is valid before it is considered to be too faint to mark targets. Defaults to 180 (3 Minutes)
checkInRange	Distance (in meters) to the outer boundary of the WP Zone in which a player plane can check in. If, for example a WP has a radius of 5 km and check-in range is 7 km, players can check in when they are 13 km or less away from the zone’s center. Defaults to 10’000m (10 km)
ackSound	Sound file to play after a player has successfully transmitted target coordinates. Defaults to <none>
guiSound	Sound file to play whenever a player uses a menu item from the WP “FAC” menu

5.4.13.4 ME Attributes

Name	Description
wpTarget	Marks this zone as a WP Zone. The value of this attribute defines which coalition uses this zone. Valid values are • RED or 1 • BLUE or 2 **MANDATORY**
shellStrength	Explosive power of shell hitting the ground. Defaults to 500
shellNum	Number of shells per artillery salvo Defaults to 17
transitionTime	The time (in seconds) it takes for shells to arrive after the ‘fire’ command is given Defaults to 20
coolDown	Number of seconds after which the artillery can fire again. Defaults to 180 (3 minutes)
baseAccuracy	Radius (in m) around the wp where the shells hit in. Defaults to 50
method	DML method for the outputs

Name	Description
wpMethod	Defaults to “change”
wpFired!	Output that is signaled when a successfully Fire command was received by the WP zone Defaults to <none>
checkInRange	Distance (from the zone’s border) from which a plane can check into the zone. Defaults to 10km (10000) Example: if a WP zone has a radius of 3 km, and a checkInRange of 2 km, a plane can check into the zone when it’s 5km (= 3 + 2) or less away from the zone’s center. Be careful with check-in ranges when they overlap, as this may lead to confusing situations for pilots who may inadvertently sign into the wrong zone.
ackSound	Sound file to play after a player has successfully transmitted target coordinates. Overrides any setting of the wpConfig Defaults to what you defined in wpConfig
guiSound	Sound file to play whenever a player uses a menu item from the WP “FAC” menu. Overrides any setting of the wpConfig Defaults to what you defined in wpConfig

5.4.13.5 Using the module

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

Optionally, add a config zone to set module-global options.

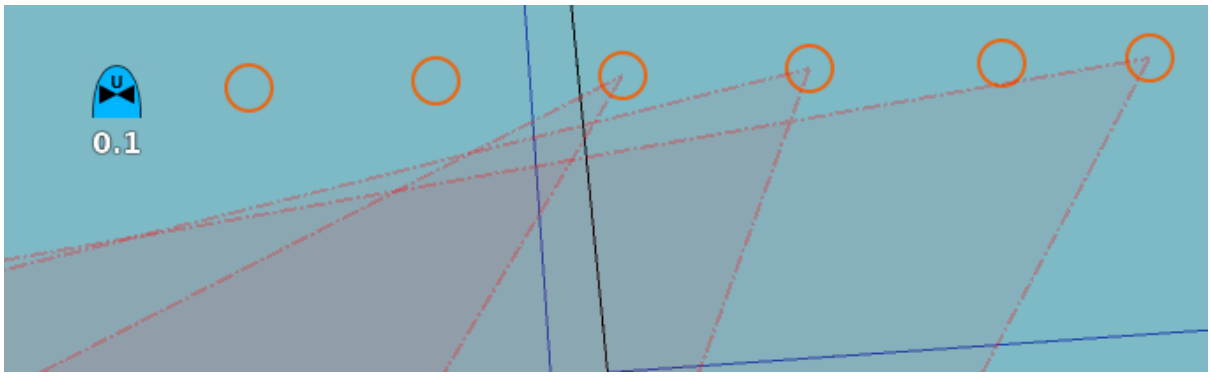
Add Zones with the wpTarget attribute, and optionally set some other attributes to control firepower, number of shells etc.

Equip at least one player plane with WP ammo.

5.4.14 ASW (Anti-Submarine Warfare / Sub Hunters)

5.4.14.1 Description

ASW are a set of modules that add an easy Anti-Submarine Warfare ability for your missions that is currently absent from DCS. The module simulates a streamlined approach to ASW: units (including player) can deploy ASW Buoys that have the ability to detect and pinpoint enemy submerged crafts. Once enough buoys detect the same enemy, they can calculate a “fix” and units can deploy ASW Torpedoes to try and destroy the submarine. Buoys, Torpedoes, and submerged contacts are tracked and drawn on the F10 map for player visualization.



Dropped buoys are also marked with smoke for the duration of their lifetime (usually 30 minutes). Buoy management, submarine detection, submarine fix and torpedo attacks are handled by the ASW module automatically.



All ASW functionality is implemented entirely virtually; it doesn’t require a particular DCS module to be installed and does not use custom models. It’s therefore compatible with all missions and maps. Using ASW on some maps (e.g., Nevada) may be questionable, but is entirely possible.

DML’s ASW package is split into multiple modules that each provide a certain ASW aspect to give you more flexibility:

- **asw** – provides management for deployed buoys and torpedoes: tracking, sub detection, sub attacks, F10 map updates, smoke.

- **aswZones** – provides DML integration of the asw module via zones that allow AI units to deploy buoys and torpedoes, and player units to stock/restock ASW munitions (buoys, torpedoes)

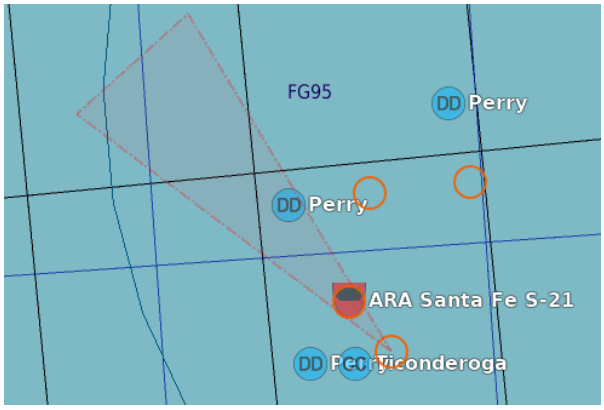
- **aswGUI** – provides a convenient Communications→Other→ASW player interface that allows player units to load ASW munitions when inside an ASW zone (integration with aswZones) and allows them to drop buoys and torpedoes

- **aswSubs** – turns nice and mostly harmless standard DCS submerged vessels into nasty, lethal attack craft that can sink even aircraft carriers when they get close. This script allows DCS submarine units to automatically destroy surface vessels without having to surface, and they will kill any enemy contact that comes too close.

ASW Buoys

Buoys are dropped by units (naval and airborne, by AI with aswZones and by player with aswGUI). They have a limited lifetime (30 minutes by default) and are marked on the F10 map with a circle and with smoke on the water (yeah, I've been aching for years to be able to write those last four words in a real context).

While alive buoys constantly scan for enemy submerged contacts. Should they detect an enemy contact, they mark the direction and a 'confidence cone' (a wedge, see right) that marks the area where the enemy sub is likely to be. Note that unlike in the image to the right, you'd normally not see the red enemy sub. That mission's 'show all' option was turned on for demonstration purposes. Normally you would turn that off to make the mission more interesting.



ASW buoys can detect enemy submarines at up to 12km, although their confidence and precision drops dramatically with range (and drops to 0 at more than 12km).

All ASW Buoy data is automatically gathered by the ASW module, and if enough buoys locate the same sub with enough precision, a fix is made, and marked on the map. Fixes, unless refreshed buy enough buoy data, live for 3 minutes, after which the data becomes stale and less reliant. A fix greatly increases an ASW torpedoes likelihood to hit a submarine. Contacts and fixes are also announced with sound effects (the ASW module comes with distinct sounds for each and defaults to sound files with that name)

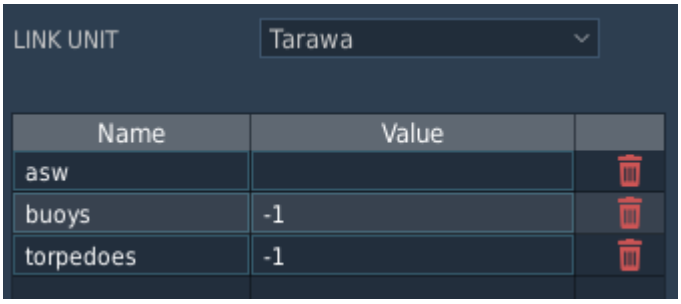
ASW Torpedoes

Units (naval and airborne, AI and players) can drop ASW torpedoes that hunt for submerged craft. Their range is more than 12km and they run at 55kts, but their likelihood to actually track and destroy a sub is small unless they get their initial targeting information from a nearby fix, or dropped very close to a submarine. Furthermore, torpedoes do not differentiate between enemy or friendly subs, so dropping a torpedo without a nearby fix in waters where friendly subs are operating is not advised. If a torpedo finds a sub, it starts homing in on it, at which time the submarine's destruction is nearly certain. During their lifetime, torpedoes are tracked on the F10 map. If a torpedo goes active and starts homing, it also broadcasts that

information to its coalition. ASW torpedoes never attack surface vessels nor submarines that have surfaced.

ASW Munitions and Stores (aswZones)
You use ASW Zones for two purposes:

- **Supply ASW Buoys and ASW Torpedoes for players**:
  If a player lands inside an aswZone, they can load up on buoys and torpedoes. You can place a limit on the number of buoys or torpedoes that are available, and if stores are limited, aswZones automatically provides stores management: players can unload ASW munitions from their aircraft, which are then added to the available stock, and vice versa.
  Note that aswZones can also be placed on land for airfields to supply ASW munitions.



- **Allow AI units to drop buoys and torpedoes**
  aswZones have inputs for "buoy?" and "torpedo?" that causes them to drop a buoy or torpedo, respectively. When you link such an aswZone to an AI (or player) unit, these units gain the ability to drop asw buoys and topedoes. Inventory tracking also applies to asw munitions dropped by "buoy?" and "torpedo?", so an aswZone can't drop more buoys nor torpedoes than it has in stock.



ASW player interaction (ASW GUI)
Since dropping ASW munitions can be added to all units by attacking an aswZone, the following applies only to the interactive player (GUI) part of ASW:

By default, only the following aircraft can pick up and drop ASW Munitions:

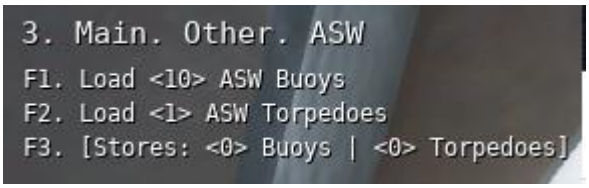
- Huey (UH-1)
- Hind (Mi-24)
- Hip (Mi-8)

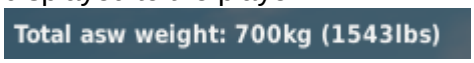
Since ASW munitions have (significant) weight, an airframe's cargo capacity is an important factor in it's ASW suitability.

That being said, ASW capabilities are by no means restricted to these three helicopters, nor helicopters in general. Access to aswGUI is controlled with aswGUI's config zone, and the aswCarrier attribute controls which aircraft are allowed access.

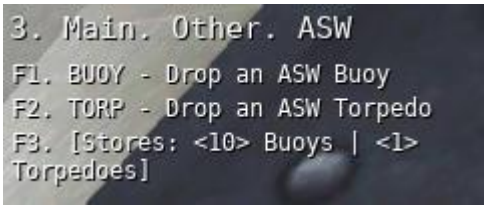
Players have access to two different interfaces to control ASW munitions:

- When landed and inside an aswZone, they can load and unload ASW stores. Each aircraft has two “slots” to load / unload ASW munitions, and each slot can either carry up to ten ASW Buoys or up to two ASW Torpedoes. Each buoy weighs 50kg (102lbs), and each torpedo 700kg (1543lbs). The interface does not prevent the player from overloading their aircraft; for example, a Huey loaded up with 4 ASW torpedoes will not make it far. Each time a player loads or unloads ASW munitions, the new total ASW weight is displayed to the player.





- When airborne, the ASW interface allows the player to drop buoys and torpedoes and automatically manages ASW stores. Note that although not helpful, ASW munitions can be dropped over land.
When ASW munitions are dropped over open water, they are automatically handed off to the main asw module and start searching for submarines.



aswSubs

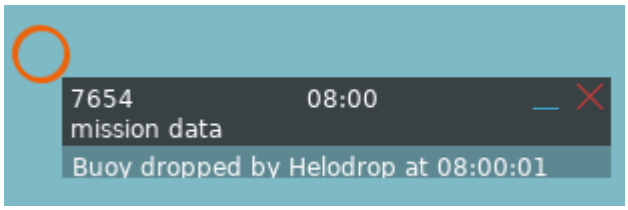
A good module needs good antagonists, and DCS’s subs currently couldn’t cut it. The aswSubs module makes bad subs *really* bad. It imbues submarines with the ability to strike without surfacing, and destroy even aircraft carriers if they get close enough. You use trigger zones to designate targets, and any enemy sub that gets close enough to any naval unit on the target list automatically attacks them. Note that aswSubs will not make units change their course, only put their attack abilities on steroids should they be able to get close enough.



Once an aswSub gets into range of a target ship, it loosens a number of torpedoes on that ship that can’t be dodged. The torpedoes inflict damage on the ship. You can configure most of these parameters with a config zone. By default, an aswSub attack isn’t survivable even for the largest of ships (aircraft carriers). Once a ship was successfully attacked by an aswSub, that ship won’t be attacked again.

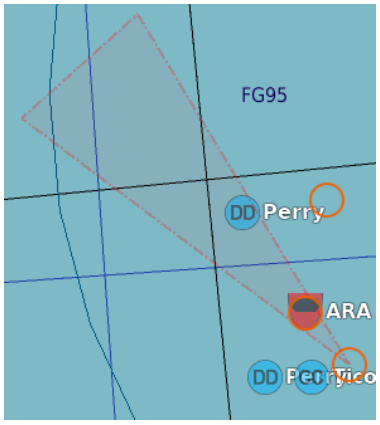
Messages, Markings and Signals
The ASW module sends out messages, map marks and audio signals (when the correct sound files are added to the mission) as follows:

- **Buoys (Map, In-game)**
  When a unit (AI or Player) drops a buoy, it is marked on the F10 map by a circle. A buoy has a limited life (30 minutes by default) and disappears after that. In-game, a buoy is marked by smoke (red by default). Buoys can be dropped over land, however, they simply disappear (no mark, no smoke)

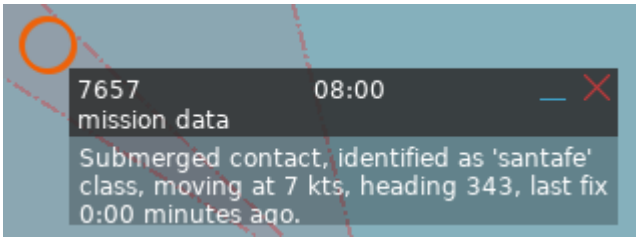


- **Contact (Map, Audio)**
  'beep-beep' audio signal and a red semi-transparent wedge emanating from a buoys. This happens when a buoy detects a sub. The audio signal defaults to the file "beacon beep-beep.ogg" which must be part of your mission (the file name can be configured).

  The red wedge represents a "90% probability cone": there is a 90% probability that a submarine is located within that cone. The bigger (wider) the wedge, the weaker the contact to the sub.
  Currently, there are no false positives implemented, so if a player sees a wedge, there definitely is a submarine around.



- **Contact Fix (Map, Audio, Messages)**
  When enough buoys are seeing the same submarine with enough confidence, a position fix can be triangulated. An audio sound ("submarine ping.ogg" by default, part of DML's library, must be included in your mission to play. File name can be configured) plays and a mark is placed on the F10 map. While the fix is active, it's position is updated. Fixes have a lifetime of 3 minutes since the last positive identification of the sub by buoys.



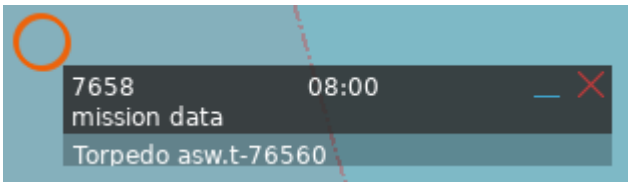
  A fix is also announced by a message like



  Whenever a fix is re-confirmed by buoy triangulation (and therefore its life timer renewed) a message appears similar to



- **Torpedoes (Map, Messages, In-game)**
  When a unit (AI or Player) drops a torpedo, it is marked on the map. The mark is updated every 10 seconds. A torpedoes' lifetime is 8 Minutes (by default) during which it can travel well over 12km. The status of a torpedo is communicated with messages such as





or



If a torpedo is homing, it will most likely (unless the enemy sub manages to outrun it) destroy the sub, which is announced by a message



and an in-game sub-surface explosion and water fountain at the location of the sub. Note: this explosion can be harmful to units directly above, including low-flying aircraft, if they are too close.

If the torpedo doesn't engage a submarine, it will eventually stop and be removed from the map with the message



Integration with PlayerScore
ASW has full integration with the PlayerScore module. To use this feature, set the attributes "killScore" and "killFeat" in ASW's configuration zone.

5.4.14.2 Dependencies
There are some dependencies between the various ASW modules

- **asw** requires dcsCommon, cfxZones.
- **aswZones** requires dcsCommon, cfxZones and **asw**
- **aswGUI** requires dcsCommon, cfxZones, **asw** and **aswZones**
- **aswSubs** requires dcsCommon, cfxZones

Optionally, asw supports playerScore, no load dependencies (can be added in any order)

5.4.14.3 Module Configuration
Most of DML's ASW capabilities can be curtailed with config zones:

5.4.14.3.1 asw (main)

The asw module mainly controls buoy and torpedo properies and behavior. You use a configuration zone to modify some behavior. To configure asw via a configuration zone,

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

Name	Description
verbose	A value of “true” turns on debugging for the entire module. Default is “false”
buoyLife	Duration (in seconds) how long a buoy is active. Defaults to 1800 (=30 minutes)
fixLife	Number of seconds that a sub fix is reliable and can be used by nearby torpedoes to get a better homing chance. Defaults to 180 (=3 minutes)
detectionRange	Sensor range (in meters) for a buoy. Reliability deteriorates with distance, and beyond this range, no subs are detected. Only enemy subs are detected when in range Does not detect vessels on the surface, even if those vessels are of the submarine class Defaults to 12000 (12km)
sureDetect	“point blank’ range for buoys. If a sub is closer than this range they are always detected. Defaults to 1000
detectionDepth	Maximum depth (in meters) in which submarines are detected. Submarines that are deeper than that are invisible and will not be detected. Defaults to 500
fixSound	Sound effect to play when a new fix on a submarine is made Defaults to "submarine ping.ogg" (file is part of DML’s library)
sonarSound	Sound effect to play when a buoy finds a contact. Defaults to "beacon beep-beep.ogg" (file is part of DML’s library)
redKill!	Output flag to bang! when a red submarine is killed by ASW means. Defaults to <none>
blueKill!	Output flag to bang! when a blue submarine is killed by ASW means. Defaults to <none>
method	Method for output flags
smokeColor	Color that marks a buoy. Can be a number (0 to 4) or “red”, “green”, “blue”, “white”, “orange.” Defaults to “red”
killScore	Only relevant when PayerScore is active in mission Score for killing a submarine with ASW munitions Defaults to 0 (no score)
killFeat	Only relevant when PlayerScore is active in mission Description of feat when killing a submarine with ASW munitions, supports all PlayerScore wildcards.
useSmoke	When set to false, dropped buoys do not emit smoke Defaults to true (buoyes also emit smoke)

5.4.14.3.2 aswZones

The aswZones module deals with ASW munition availability via Zones. You use a configuration zone to modify some very limited behavior. To configure aswZones via a configuration zone,

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

Name	Description
verbose	A value of “true” turns on debugging for the entire module. Default is “false”

5.4.14.3.3 aswGUI

The aswGUI module manages player interaction with the ASW module, including player aircraft loadout and which player aircraft are eligible for ASW. You use a configuration zone to modify some behavior. To configure aswGUI via a configuration zone,

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

Name	Description
verbose	A value of “true” turns on debugging for the entire module. Default is “false”
aswCarriers	Comma-separated list of types of player aircraft that are allowed to perform ASW. Defaults to DCS Common’s list of troop carriers (`Mi-8MT, UH-1H, Mi-24P`), and you can provide your own list: Example: “`Mi-8MT, UH-1H, C-130J`” removes the Hind and adds the Hercules fixed-wing transport to the list of legal ASW carft. Supports wildcard type endings: if a type ends on an asterisk (“*”) all types that match whatever precedes the asterisk are accepted. For example, “`Mi-*`” will match both “`Mi-8T`” and “`Mi-24P`”. You can supply the type ‘`any`’ or ‘`all`’ to allow all aircraft to perform ASW. Default <none> (use dcsCommon’s list of troop carriers for ASW)
buoyWeight	Weight (in kg) per ASW buoy Defaults to 50kg
torpedoWeight	Weight (in kg) per ASW torpedo Defaults to 700kg

5.4.14.3.4 aswSubs

You can change key aspects of how aswSubs attack enemy surface vessels by providing the correct attributes in a config zone. To configure aswSubs via a configuration zone,

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

Name	Description
verbose	A value of “true” turns on debugging for the entire module. Default is “false”
critDist	Distance (in meters) to trigger the submarine’s attack on a target vessel (see *targets*, below). The target vessel will receive multiple (see *salvoSize*) hits (see *explosionDamage,* below)
explosionDamage	Damage inflicted by attacking aswSub per hit. Defaults to 1000
salvoSize	A range (e.g., 2-4) that determines how many hits a target receives when attacked. Once a target ship has been hit, no other aswSub will attack that vessel. Defaults to ‘4-4’ (always take four hits)
targets	Comma-separated list of group names that are targets for aswSubs. The group must exist in ME when the mission starts. Defaults to <none>

5.4.14.4 ME Attributes

5.4.14.4.1 asw (main)

(no ME integration)

5.4.14.4.2 aswZones

Name	Description
asw	Marks this zone as an ASW supply/drop zone **MANDATORY**
buoys	Number of buoys in stock in this zone. Setting this number to -1 sets the supply to infinite. Defaults to -1 (infinite supply)
torpedoes	Number of torpedoes in stock in this zone. Setting this number to -1 sets the supply to infinite. Defaults to -1 (infinite supply)
coalition	Coalition which owns the asw that are being dropped by this zone. If this zone is linked to a unit, that unit’s coalition is used instead of any value given to this attribute. (only required when dropping ASW and zone is not linked to a unit) Defaults to ‘neutral’
buoy?	Input flag that triggers the drop of an ASW buoy. Will only drop a buoy if there are enough left in the stores. Defaults to <none>
torpedo?	Input flag that triggers the drop of an ASW torpedo. Will only drop a torpedo if there are enough left in stores. Defaults to <none>
triggerMethod aswTriggerMethod	DML Method for input flags Defaults to ‘change’

5.4.14.4.3 aswGUI

(no ME integration)

5.4.14.4.4 aswSubs

(no ME integration)

5.4.14.5 Using the module

Add the main asw module to your mission. You now have ASW capabilities. To

- be able to drop ASW munitions from AI-controlled craft, add the aswZones module, and attach trigger zones to the units that should be able to drop ASW munitions. Use their buoy? and torpedo? inputs to trigger drops
- be able to control ASW aircraft, add the aswZones and aswGUI modules. Use aswZones to set up zones where players can load up asw munitions
- have submarines attack enemy ships while submerged, add the aswSubs module and set up target groups via its config zone

5.4.15 Taxi Police

5.4.15.1 Description

Taxi Police is a module to enforce a speed limit on the taxiways of airfields.

IMPORTANT!
When you are using this module, you are enforcing a realistic, albeit (for most game setings) unnecessary rule, and *many players might not take well to this*. You may want to think twice before you include this module in your mission, lest you lose players who become annoyed with the restriction that this module imposes.



While imposing a speed limit may not be the best of ideas for most missions (players want to have fun, not follow some arbitrary rules – they have the real world for that), so missions may require a modicum of policing (some of) their airfield's taxiways.

FEATURES

TaxiPolice monitors all player fixed-wing aircraft that are on the ground and close to an airfield for speeding violations. Aircraft on ships or FARPs are *not* monitored, neither are helicopters.

If a monitored aircraft moves faster than the current map-wide speed limit (50 km/h or 27 knots by default), and that aircraft is not on a runway, the player is put on notice, and a taxi way offense is registered against them.

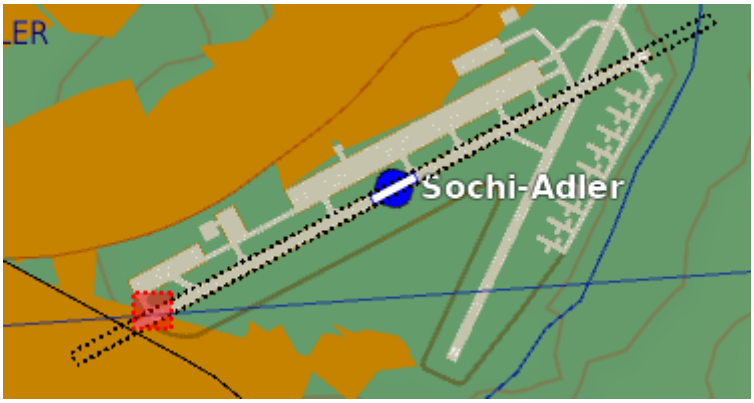
If the player commits repeated offenses, once the number of violations exceeds the limit (3 by default), every time they transgress, their aircraft is rendered unable to depart (taxiPolice adds 200 tons of cargo), requiring the player to re-slot. Furthermore, all other players are alerted to this player's reckless and asocial behavior. Transgressions are registered against the pilot, not aircraft, so if the same pilot commits further violations they'll be sanctioned even after changing aircraft and coalition. All transgressions are forgiven only after the mission restarts.

TaxiPolice is fully automatic and automatically detects all runways on each map (see limitations, below). When active, the speed limit is enforced on all airfields on the map, unless you actively exclude an airfield through a trigger zone

TaxiPolice can be enabled and disabled dynamically while the mission is running.

LIMITATIONS

TaxiPolice relies on the airfield information it receives from the map. Not all maps return all runways correctly. For example, Caucasus' Sochi-Adler airfield sports two separate runways:



However, DCS currently only reports one of them (the one with the black dotted outline). Should a player try to depart from the unrecognized runway, taxiPolice will “fine” them (record a violation the first time, and eventually, on their fourth infraction, keep them on the ground).

When in doubt, turn on taxiPolice’s verbosity, and look at the F10 in-game map. All recognized runways are marked by a dotted black outline. Speeding **outside** those dotted areas will be policed.

5.4.15.2 Dependencies
TaxiPolice requires dcsCommon and cfxZones

5.4.15.3 Module Configuration
Since most of TaxiPolice’s rules apply globally, nearly all its functionality is controlled with its configuration zone.

To configure TaxiPolice via a configuration zone,

- Place a Trigger Zone anywhere on the map inME
- Name it “taxiPoliceConfig” (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”
speedLimit	The speed limit for aircraft on all taxiways and tarmacs, set in m/s (meter per second, a.k.a. ‘civilized units’). Recommended values are 10 (36 km/h) and 14 (50 km/h) Defaults to 14 (14m/s = 50 km/h = 27 knots)
triggerTime	Grace period (in second) after which a transgression is registered. If a pilot goes above the speed limit for less than that time, it is forgotten. Defaults to 3 (seconds)
leeway	The width in meters of the ‘corridor’ along a runway to which the runway ‘no limit’ bounds extend to each side. Thus, exceeding the speed limit just off the runway is allowed – in case the pilots veers off the runway slightly. To see each airfield’s runway corridors, turn on verbosity, and switch to the F10 Map view. All runway corridors are marked on the

Name	Description
	map with a dotted black line Defaults to 5 (meters)
extend	The length in meters of the 'corridor' along the runway to with it extends in front of and behind the runway. That way, pilots who touch down too early or too late (and survive) don't get in more trouble than they already are. To see each airfield's runway corridors, turn on verbosity, and switch to the F10 Map view. All runway corridors are marked on the map with a dotted black line. Defaults to 500 (meters)
radius	The radius (in meters) around an airfield's location as defined in DCS's airfield data base. Aircraft inside this radius come under scrutiny if they are on the ground. To see each airfield's center, turn on verbosity, and switch to the F10 Map view. Each airfield's center is marked on the map with a dotted red line. Note that in many maps (e.g., Caucasus), that airfield's location coincides with the runup area of a particular runway, not the center of an airfield as one would have naively assumed. Defaults to 3000 (meters)
maxTickets	Number of tickets a pilot (not unit) may receive before TaxiPolice starts taking active retribution for offenses. Defaults to 3 (i.e. the fourth offense will be painful)
active	Initial state of taxiPolice when the mission starts up. When set to false, TaxiPolice is off duty until enabled. Defaults to true (taxiPolice is active at mission start)
greetings	When set to true, TaxiPolice will greet every pilot upon slotting or landing on an airfield, unless they are off duty. Pilots are given the current speed limit in km/h and knots. If an airfield is exempt from policing (as set with a 'taxiPolice' attribute at an airfield), that fact is also mentioned. Defaults to true – pilots are updated of the current speed limit and enforcement policies of their current airfield
onPatrol	Watchflag. Whenever the value of this flag changes, TaxiPolice becomes active on all policed airfields. A NOTAM is sent to all current players. Player's transgressions are recorded and sanctioned when they exceed the maxTicket limit. Note that turning taxiPolice on or off does not reset a player's violation count, only restarting the mission will do that. Defaults to <none>
offDuty	Watchflag. Whenever the value of this flag changes, TaxiPolice steps off the plate and ingnores any speed limit violations on the map. A NOTAM explaining that fact is sent to all current players. Note that turning taxiPolice on or off does not reset a player's violation count, only restarting the mission will do that. Defaults to <none>

5.4.15.4 ME Attributes

By default, TaxiPolice manages all fixed airfields on the map. Occasionally, you may want to exclude an airfield from policing. In ME you can add trigger zones to airfields so TaxiPolice ignores them. To do so, create a trigger zone close to the airfield, and add the following attribute:

Name	Description
TaxiPolice	When present, **the airfield closest to this trigger zone** is removed from TaxiPolice's list of airfields to monitor. Speeding on taxiways, ramps or tarmac areas is ignored. **MANDATORY**

5.4.15.5 Using the module

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

Optionally, add a config zone with ME to curtail TaxiPolice's rules, and use trigger zones to exclude some airfields from policing.

5.4.16 Shallows (automatically remove ship husks)

5.4.16.1 Description

DCS currently has an obscure bug where ships, after they are destroyed, can't sink when the water is too shallow. If there is no 'destroyed' version of the ship, the hull remains in shallow waters, and looks undamaged to pilots who may then commence to expend ammunition on a dead target.





Shallows is a script that first visually marks a dead ship with fire and smoke, and after 5 minutes removes the hull.

5.4.16.2 Dependencies

Shallows is stand-alone and does not require any other modules

5.4.16.3 Module Configuration

Shallows is self-contained and does not need configuration

5.4.16.4 ME Attributes

None. It just works.

5.4.16.5 Using the module

Run the shallows script from the mission before the first ship is destroyed.

5.4.17 StopGap and StopGapGUI

5.4.17.1 Description

StopGap is a module that fills unused player slots with static aircraft – until a player enters that slot, at which point the static aircraft is replaced with the 'real' player aircraft.



The result is that in missions with many possible player aircraft, the airfields no longer look deserted. The downside: if you add many different aircraft slots, your performance may suffer – just like it would if you added all those planes as static objects by hand.

USING STOPGAP

StopGap simply works. That doesn't mean that there aren't a few recommendations that help mission designers:

- Place player planes/helicopters with 'From Ground' orders (hot or cold). That way you can control their orientation. **NEVER** use "From Parking" or "From Ramp" with stopGap (make sure to fence out these aircraft with a trigger zone, see below)
- Don't place player planes with 'From Ground' orders inside shelters, or they will be locked in (a DCS bug)
- Whenever possible, use single-unit player groups (multi-aircraft groups are supported)
- Carrier-based aircraft are not supported (stopgap ignores those).
- Be careful when placing the Harrier. For reasons unknown, DCS likes to drop it like it's hot. From altitude.

All that is required is that you add the stopgap module to your mission.

EXCLUDING PLAYER AIRCRAFT FROM STOPGAP

Sometimes you may want to tone down the eye candy, or even hide the fact that player-controlled aircraft are available at an airfield

To exclude showing player aircraft,

- use a triggerZone with a *stopGap* = *false* attribute (preferred method) and player aircraft inside the zone are not replaced with statics

- use a triggerZone with a stopGapSP = false attribute, and player aircraft inside this zone are not replaced with static aircraft in single-player, but will be replaced with statics in multiplayer mode (requires stopGapGUI running on the server)
- you can also add '-sg' to a player unit's or group's name to exclude that unit/group from stopGap. I consider naming schemes to be lazy and uncouth design. But they work, as the stand-alone version shows.
- You can add "-sp' to a player unit's group or unit name to exclude that unit/group from being replaced by statics in single player mode only (it will be replaced in multiplayer mode, provided that stopGapGUI is running on the server)

To hide player aircraft when the airfield belongs to another coalition,

- use the cfxSSBClient module which also automatically blocks player planes when they are on a hostile airfield. stopGap automatically integrates with SSB when it's present.

Multiplayer STOPGAP: StopGapGUI and Slotty

StopGap supports multiplayer – in fact it runs much better in Multiplayer than Singleplayer. Some peculiarities in DCS require that for multiplayer, the tiny server script "stopGapGUI" runs on the server (only the server is required). This script helps all clients to correctly synchronize their actions so that the static stand-ins and real player planes do not collide.

When playing a StopGap-enabled mission in multiplayer, StopGap automatically detects and respects SSB if that tool is running. Blocked slots will then appear empty.

Instead of installing SSB on the server, you can choose to add the 'slotty' module to your mission. This removes the *requirement* to install SSB on the server, yet still allows server admins to install SSB – slotty and SSB cooperate nicely.

Installing StopGapGUI

Copy the "stopGapGUI.lua" file from DML's "server modules" folder into the

```
(user)/Saved Games/DCS/Scripts/Hooks/
```

folder. Then restart DCS (files in /Hooks only load when DCS re-starts). You are done, from now on any mission with StopGap is served correctly.

FEATURES

There are two versions of StopGap available: a stand-alone version, and the DML one. The difference between those two versions is that, having access to DML, the DML version adds several features that stand-alone doesn't have:

Features shared among versions:

- Automatically supports SSB ("Simple Slot Block" a popular player slot blocking server extension) when it is active: a blocked slot will not show its static aircraft
- Exclude planes or groups by adding '-sg' to the name of unit/group

DML-Only

- Turn on / off at will using flags. Use this to allow players to choose if they wish to use StopGap or not. Note that this is a mission-global setting: it applies to all airfields, across all clients (when in multiplayer).
- You can optionally turn off SSB detection. In that case, blocked slots are filled with their stand-ins.
- You can exclude aircraft from stopGap by placing them in zones that have a *stopGap = false* attribute or *stopGapSP=false* attribute.

See also the "*SittingDucks*" module (later in this document) for a tactical extension of the StopGaps functionality: making player slots not only visible on the ground, but *vulnerable*.

5.4.17.2 Dependencies
StopGaps requires dcsCommon, cfxZones and cfxMX to run.

stopGapGUI has no dependencies, must be installed in /Hooks of the server (only server required).

Note

The 'slotty' module (as part of a *mission*) can achieve the same results as stopGapGUI without requiring a server module to run.

5.4.17.3 Module Configuration
stopGaps works out-of-the-box, and supports a number of QoL features that you can customize via a config zone.

To configure stopgap.,

- Place a Trigger Zone anywhere on the map
- Name it "stopGapConfig" (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"
onStart	Controls if stopGap is on when the mission starts up. Defauts to "true" – stopGap is enabled
on?	Watchflag. A trigger on this flag turns an inactive stopGap to active (all unoccupied player parking slots are filled with aircraft) Defaults to <none>
off?	Watchflag. A trigger on this turns stopGap off, and removes all static aircraft that it placed on player slots. Defaults to <none>
triggerMethod	DML Method to determine when a Watchflag should trigger Defaults to "change"
ssb	(Multiplayer only) Controls if stopGap automatically interfaces with server-based SSB (Simple Slot Block) module, when present. If a slot is blocked,

Name	Description
	stopGap will not place an aircraft there, leaving the corresponding parking slot free. If disabled (set to false), blocked slots are filled with static stand-ins. Defaults to true (interface with SSB) **Note: if SSB isn't installed on the server, this option does nothing.**
refresh	Interval (in seconds, e.g. 3600 is one hour) in which all static replacements are refreshed. Only useful on long-running server missions as a purely eye-candy option (regularly all static replacements are reset, so they look fresh and all visual blemishes (bomb damage etc.) are removed. A negative value turns this option off **WARNING:** Do not use in conjunction with sitting ducks, as this option does the opposite of what sitting ducks achieves. Defaults to -1 (disabled)
kickDead	In multiplayer, if a player dies, they are kicked from their aircraft to spectators. This prevents a rare DCS bug from occurring (that mostly affects helicopters) where re-spawning players crash in their newly re-slotted plane. Note that kickDead = true (the default) can interfere with autoCSAR if you are flying warbirds in MP (the symptoms being that no CSAR mission is auto-generated for players leaving warbirds). If you run into this issue, set kickDead to false. Defaults to true (dead players are "kicked" to spectators)
allNeutral	If set to true, all stand-in statics belong to the NEUTRAL faction instead of the slot that the player's plane belongs to. This allows the static stand-ins to appear inside the game without the risk of being attacked. Defaults to false (all stand-ins belong to the same faction as their player slot)
noParking	If set to true, player aircraft that start from Parking (hot or cold) are ignored. Defaults to false

5.4.17.4 ME Attributes

Name	Description
stopGap	When the value of this attribute is **false**, any player aircraft inside this zone will not receive a static stand-in, their slot remains empty. Defaults to true (unit receives a static stand-in) **MANDATORY**

Alternatively, you can disable StopGap only when the mission is run in single-player mode (this is useful when, due to some map properties, the aircraft ‘falls out of the sky’ in single-player, but works well in multiplayer sue to StopGapGUI’s superior synching.

Name	Description
stopGapSP	When the value of this attribute is **false**, any player aircraft inside this zone will not receive a static stand-in **in single-player mode**, their slot remains empty. In multiplayer mode, the slot is filled with a static stand-in. Requires stopGapGui on the server or the slotty module in the mission itself. Defaults to true (unit receives a static stand-in) **MANDATORY**

5.4.17.5 Using the module
Add the stopGap module to your mission.

When serving a multiplayer mission, install the stopGapGUI script in the server’s /Hooks folder.

5.4.18 Sitting Ducks

5.4.18.1 Description

Sitting Ducks is a logical extension of ‘Stop Gaps”: while Stop Gaps gives a visual representation to player aircraft (“slots”) on the airfield, SittingDucks allows the mission designer to make those player slots vulnerable to enemy fire: if the plane is destroyed on the airfield, the corresponding player slot is blocked.





This allows for new strategies and tactics for your multiplayer missions, since players now have to defend their base to retain access to their planes, and can make it advantageous to attack enemy player bases to deny them theirs.

FEATURES

Can be turned on and off

Sitting ducks can be turned on and off at will. When turned off, destroying an empty player plane (i.e. the slot representation on the airfield) will no longer block that plane.

If a resupply time (see below) was set for a destroyed player plane,

Turning off sittingDucks has no effect on resupply time, neither will it re-enable blocked slots. It will merely prevent further slots from becoming blocked when their slot representations are destroyed.

Resupply

Since sittingDucks is most likely used in long multiplayer engagements (i.e. running on servers for long periods of time, with a changing base of players over mission runtime), sittingDucks provides an automatic ‘resupply’ option that makes destroyed player planes available after some time to mimic supplies being delivered.

5.4.18.2 Dependencies
Sitting Ducks requires

- The mission to be played as multiplayer
- SSB installed on the server (only server), configured with `SSB.kickReset = false`
- StopGapsGUI installed on the server (only server)

SittingDucks requires dcsCommon, cfxZones and stopGaps to run.

5.4.18.3 Module Configuration
sittingDucks works out-of-the-box, provided that the required other modules are installed and the hosting server also runs the required modules. It supports the following options.

To configure sittingDucks,

- Place a Trigger Zone anywhere on the map inME
- Name it “stopGapConfig” (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. Defaults to “false”
onStart	Determines if the mission starts with sittingDucks enabled or turned off. Defaults to ‘true’ (enabled on mission start)
on?	Watchflag to turn sittingDucks on (enable it) when it is off Defaults to <none>
off?	Watchflag to turn sittingDucks off (disable it) when it is enabled Defaults to <none>
triggerMethod	DML Method to determine if a Watchflag should trigger Defaults to “change”
resupplyTime	Time (in seconds) until a player is re-opened after the static stand-in was destroyed. If you supply -1, the slot remains closed indefinitely. Defaults to -1 (indefinitely closed)

5.4.18.4 ME Attributes
sittingDucks has no active zones.

5.4.18.5 Using the module
Add the sittingDucks module to your mission.

5.4.19 NoGap and NoGapGUI

5.4.19.1 Description

NoGap is StopGap's sibling. It does essentially the same, and works slightly different, with some lesser abilities when it comes to working with popular scripts like SSB (simple slot blocker).

Like StopGap/NoGapGUI, NoGap/NoGapGUI replaces inactive player aircraft with static replacement to make airfields look better.

The difference only comes into play when you (the mission designer) create player groups that contain more than one player aircraft: NoGap deals with player aircraft individually. With StopGap, when the first player slots into an aircraft in a group of multiple player aircrafts, all static replacements of that group vanish. This is because stopGap deals with player aircraft on group basis, for compatibility with popular scripts like SSB. NoGap, on the other hand, only removes the player plane's static standin, and leaves the other stand-in until other players slot in.

The downside is that NoGap isn't compatible with SSB, sittingDucks and other scripts that equally deal with player aircraft on a group basis.

USING NOGAP

NoGap works like StopGap, with similar recommendations, bar one:

- Place player planes/helicopters with 'From Ground' orders (hot or cold). That way you can control their orientation.
- Don't place player planes with 'From Ground' orders inside shelters, or they will be locked in (a DCS bug)
- **Unlike StopGap, you can use multiple player aircraft in the same group.** Due to DCS's many idiosyncrasies it's still not *recommended*, though…
- Carrier-based aircraft are not supported (stopgap ignores those).
- Be careful when placing the Harrier and Helicopters. For reasons unknown, DCS likes to drop them from height in single player mode. Use the 'exclude from singleplayer' option in this cases.

To use, all that is required is that you add the noGap module to your mission.

EXCLUDING PLAYER AIRCRAFT FROM NOGAP

Sometimes you may want to tone down the eye candy, or even hide the fact that player-controlled aircraft are available at an airfield

To not show player aircraft,

- use a triggerZone with a *noGap* = *false* attribute (preferred method) and player aircraft inside the zone are not replaced with statics
- use a triggerZone with a noGapSP = false attribute, and player aircraft inside this zone are not replaced with static aircraft in single-player, but will be replaced with statics in multiplayer mode (requires noGapGUI running on the server)
- you can also add '-ng' to a player unit's or group's name to exclude that unit/group from noGap. I consider naming schemes to be lazy and uncouth design. But they work, as the stand-alone version shows.

- You can add “-sp’ to a player unit’s group or unit name to exclude that unit/group from being replaced by statics in single player mode only (it will be replaced in multiplayer mode, provided that noGapGUI is running on the server)

To hide player aircraft when the airfield belongs to another coalition,

- switch to using stopGap instead, as **noGap does not support SSB**

Multiplayer NOGAP: noGapGUI
NoGap supports multiplayer – in fact it runs even better in multi-player than single-player. Some peculiarities in DCS require that for multiplayer, the tiny server script “noGapGUI” runs on the server (only the server is required). This script helps all clients to correctly synchronize their actions so that the static stand-ins and real player planes do not collide.

Installing noGapGUI
Copy the “noGapGUI.lua” file from DML’s “server modules” folder into the

```
(user)/Saved Games/DCS/Scripts/Hooks/
```

folder. Then restart DCS (files in /Hooks only load when DCS re-starts). You are done, from now on any mission with noGap is served correctly.

FEATURES
NoGap is DML-only and supports the following features, just like StopGap:

- Turn on / off at will using flags. Use this to allow players to choose if they wish to use NoGap or not. Note that this is a mission-global setting: it applies to all airfields, across all clients (when in multiplayer).
- You can exclude aircraft from stopGap by placing them in zones that have a “*noGap = false”* or “noGapSP=false” attribute.

5.4.19.2 Dependencies
noGap requires dcsCommon, cfxZones and cfxMX to run.

noGapGUI has no dependencies, must be installed in /Hooks of the server (only server required).

5.4.19.3 Module Configuration
noGaps works out-of-the-box, and supports a number of QoL features that you can customize via a config zone.

To configure noGap.,

- Place a Trigger Zone anywhere on the map in ME
- Name it “noGapConfig” (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”
onStart	Controls if noGap is turned on when the mission starts up. Defauts to “true” – noGap is enabled
on?	Watchflag. A trigger on this flag turns an inactive noGap to active (all unoccupied player parking slots are filled with aircraft) Defaults to <none>
off?	Watchflag. A trigger on this turns noGap off, and removes all static aircraft that it placed on player slots. Defaults to <none>
triggerMethod	DML Method to determine when a Watchflag should trigger. Defaults to “change”
timeOut	If a player intends to slot in to an aircraft, its static replacement disappears in preparation of the player slotting in. In server environments where players can change their minds, this can eventually lead to noticeable gaps. To counteract this, you can choose to have the static replacements re-appear after a delay. timeOut specifies this delay in seconds. Choose a reasonable amount, e.g., 5*60 = 300 seconds = 5 Minutes. A value of 0 or smaller disables this feature. Defaults to 0 (no time out)
kickDead	In multiplayer, if a player dies, they are kicked from their aircraft to spectators. This prevents a rare DCS bug from occurring (that mostly affects helicopters) where re-spawning players crash in their newly re-slotted plane. Note that kickDead = true (the default) can interfere with autoCSAR if you are flying warbirds in MP (the symptoms being that no CSAR mission is auto-generated for players leaving warbirds). If you run into this issue, set kickDead to false. Defaults to true (dead players are “kicked” to spectators)
allNeutral	If set to true, all stand-in statics belong to the NEUTRAL faction instead of the slot that the player’s plane belongs to. This allows the static stand-ins to appear inside the game without the risk of being attacked. Defaults to false (all stand-ins belong to the same faction as their player slot)
noParking	If set to true, player aircraft that start from Parking (hot or cold) are ignored. Defaults to false
refresh	Interval (in seconds, e.g. 3600 is one hour) in which all static replacements are refreshed. Only useful on long-running server missions as a purely eye-candy option (regularly all static replacements are reset, so they look fresh and all visual blemishes (bomb damage etc.) are removed. A negative value turns this option off **WARNING:** Do not use in conjunction with sitting ducks, as this option does the opposite of what sitting ducks achieves.

Name	Description
	Defaults to -1 (disabled)

5.4.19.4 ME Attributes

Name	Description
noGap	When the value of this attribute is **false**, any player aircraft inside this zone will not receive a static stand-in, their slot remains empty. Defaults to true (unit receives a static stand-in) **MANDATORY**

Alternatively, you can disable noGap only when the mission is run in single-player mode (this is useful when, due to some map properties, the aircraft 'falls out of the sky' in single-player but works well in multiplayer sue to noGapGUI's superior synching.

Name	Description
noGapSP	When the value of this attribute is **false**, any player aircraft inside this zone will not receive a static stand-in **in single-player mode**, their slot remains empty. In multiplayer mode, the slot is filled with a static stand-in. Requires NoGapGui on the server. Defaults to true (unit receives a static stand-in) **MANDATORY**

5.4.19.5 Using the module
Add the noGap module to your mission.

When serving a multiplayer mission, install the noGapGUI script in the server's /Hooks older.

5.4.20 Bomb Range

5.4.20.1 Description

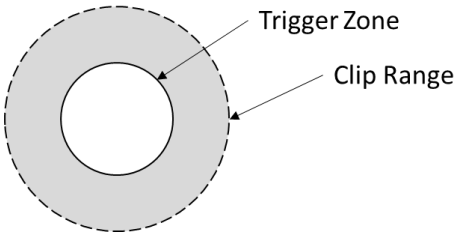
BombRange is a fully function plug-in module to add bomb range training abilities to your mission: it provides you with the ability to designate zones where player can drop munitions into, and the module reports (and keeps books) on how accurately these munitions where dropped. The module comes with a built-in UI.



BASIC FUNCTIONALITY

The module calculates impact points of player-dropped munitions (all munitions except projectiles fired by a gun) and then correlates the impact point with the closest "BombRange' trigger zone.

If the distance to the trigger zone is greater than that zone's clipping distance, the drop is ignored: all drops that fall outside the clipping distance don't count. If the projectile hits inside the clipping distance, but outside the trigger zone ("target zone"), the drop counts as a full miss ("0%"), if it impacts inside the trigger zone, the drop counts as a hit.



TRACKING HITS (ACCURACY)

Since DCS itself does not provide means to provide the impact point of munitions dropped, the module uses its own method to track hits:

- When a player releases ordnance, the module initiates tracking, and samples the weapon's position and velocity vector 20 times a second.
- When the weapon disappears in the next sample, the module assumes that the projectile has impacted in the time between now (current sample) and the last sample. It then interpolates the impact point using the last known velocity vector, height above ground, and position, assuming that the impact happened half-time between frames. The calculated impact point is then used to derive accuracy, with 100% being dead center of the target zone, and 0% being on the outside rim, or fully outside the target zone.
- This means that the average error is less than half the distance than the projectile can move between two frames (1/20th of a second by default) and inclination towards the ground: the steeper the inclination, the lower the error (most bombs tend to dive towards the ground). For most bombs, the average expected error is small (a bomb traveling at 450 knots, and with a 60° inclination towards the ground, the expected average error is less than 3 feet (1 meter). High-Velocity rockets that travel at 600 knots and have a 45% inclination can expect an average error of 9 feet (3 meters)

- If a weapon scores a direct hit on any object inside the target zone (trigger zone) the module detects this and awards 100% accuracy.

As soon as the module registers a hit close enough to (or inside) a target zone, it displays information about the impact



And then proceeds to rate the hit/miss



TARGET ZONES
Bomb Range supports circular and quad based target zones that have different abilities:

- *Circular* target zones can provide accuracy percentage based on the distance to the zone's center and target zone size. Use these to set up target zones that are intended to help people train putting bombs as close to a center as possible

- *Quad-based* target zones only provide hit/miss information: the projectile either hits inside the zone, or it misses the zone entirely. These target zones are best suited to closely surround (box in, see image on the right) vehicles or buildings where precision is tantamount, and being even slightly off is a miss.



SIGNAL OUTPUT ON FLAGS (OPTIONAL ME INTEGRATION)
Target zones provide a 'hit' attribute that can signals to flags whenever a projectile hits inside the target zone. You can use this to trigger additional events on your map should a player hit a target zone.

Globally, bomb range can also send a signal on the flags that you specify whenever a player checks in or checks out via the communications menu. This can be useful to wake up and close down activities around a bomb range for a more immersive experience.

VISUAL REPRESENTATION (OPTIONAL)
Target zones can automatically mark their boundaries with objects for increased visibility. When enabled, target zones place objects along the border, and optionally at dead center. By default, these objects are black tires with a red flag to mark the boundary, and an armed watchtower to mark the center. All demarcation objects belong to the neutral faction and hitting them has no consequences.

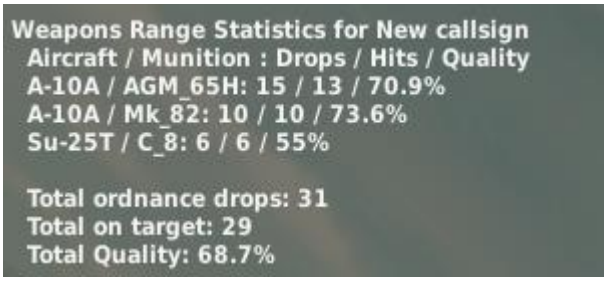
Additionally, you can have the module mark the impact locations of bombs on the ground (with red flags by default)

PLAYER ONLY
The bomb range module only considers player-controlled aircraft and ignores all ordnance dropped by AI.

UI: STATISTICS / SIGNING IN
BombRange provides a comprehensive in-game UI to all players via the 'Communications→Other…' menu. Through the interface, players can check in and out (optional), display or clear their current statistics.

- *Signing in/out (optional)*
  When this mechanic is enabled (off by default), the module only considers those weapons that are dropped by players who are checked in. This is mainly intended for missions that provide more interactive elements than a bombing range. Weapon releases from players that aren't checked in don't count even if they hit inside a target zone. If checking in is not required (a config setting), this menu will not appear.

- *Displaying Stats*
  Players can at any time display their bomb statistics. They are grouped by airframe and weapon type, so a player who switches airframes between bombing runs will see their delivery statistics for all airframes, and a grand total.



  The statistics are presented in the following format:
  Aircraft / Munition : Drops / Hits / Quality, so if a player flying a Frogfoot (Su-25T) fired 6 C-8 Rockets into a target zone, all landed inside and received an average quality of 55%, that line would look like this

```
Su-25T / C_8: 6 / 6 / 55%
```

- Resetting / Clearing Stats
  Players can reset their statistics at any time. Doing so will wipe all accumulated records and allow them to start fresh.

By default, BombRange's UI is added to the F10 Other communication menu. You can change that with the configuration zones' "attachTo:" attribute

PERSISTENCE
Bomb range supports persistence. When enabled (see persistence documentation), all player statistics are saved and restored with all other data.

LIMITATIONS
Due to the way that BombRange works, cluster munitions aren't compatible with the way that the module calculates impact points. They disappear far above the ground, and their sub-munitions usually do not hit where the bomb's trajectory would take it.

To provide better (more reliable) results for calculated impact points, Bomb Range automatically filters hit notifications for objects that are used to visually identify hits and target zones. These types are by default a red flag, a tire with a red flag, and the armed watchtower. If you are using these objects as targets, you can change the config attribute 'filterType' to prevent target filtering.

PERFORMANCE CONSIDERATIONS

Player-dropped ordnance is tracked 20 times per seconds (by default). Bomb Range uses a smart, on-demand tracking mechanism that only engages when there is ordnance to track, and limits itself only to that ordnance Therefore it has a negligible performance footprint. If it becomes an issue, you can trade performance for some accuracy (by reducing the sample time to 1/10th of a second rather than 1/20th)

5.4.20.2 Dependencies

BombRange requires dcsCommon and cfxZones.

Optional (when using "attachTo:" in the config zone): radioMenus

5.4.20.3 Module Configuration

To configure aspects of BombRange via a configuration zone,

- Place a Trigger Zone anywhere on the map
- Name it "bombRangeConfig" (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"
filterType	The names of object types, separated by comma, that never register a hit. This is used by bombRange to ignore hits generated by eye candy like flags, tires or watchtowers. Defaults to "house2arm, Black_Tyre_RF, Red_Flag".
reportLongMisses	When set to true, impacts of weapons are always reported, even if they fall outside if the closest bomb zone's clipDist. Defaults to false (impacts are only reported if they fall within the nearest bomb zone's clipDist)
mustCheckIn	If set to true, players who want their ordnance drops to be rated must check in via communications first. Defauts to false (no check-in required)
ups	Samples per second when the module monitors munitions that have been released. Note that increasing this number can significantly impact performance but give little to no additional precision. Defaults to 20, meaning that the module samples all munitions that currently have been releaset every 1/20th (0.05) seconds.

Name	Description
menuTitle	Name of the menu that is installed every player's communication→Other… menu to access the modules UI. Defaults to "Contact BOMB RANGE"
signIn!	Output flag to bang! when a player signs in to the bomb range via communications. Can be used to 'wake up' a bomb range Defaults to <none>
signOut!	Output flag to bang! when a player signs ouf the bomb range via communications. Defaults to <none>
method	Method for output flags, Defaults to "inc"
attachTo:	When present, the BombRange UI attaches to the trigger zone named as value. This trigger zone must have a "radioMainMenu" attribute. Note that if you use this attribute, you must also add the radioMenus module to your mission and that module **must** load before BombRange. Defaults to <none> (UI installs at F10 Other level)

5.4.20.4 ME Attributes

Name	Description
bombRange	Marks this zone as a bombRange target zone. **MANDATORY**
percentage	If set to true, the target zone reports in percent how close to the center the impact was located. 100% means bullseye, 0% means outside. If set to false, any hit inside counts as 100% hit. Only works for circular target zones. If you try to enable percentage for quad-based target zones, you will receive a warning at mission start, and the zone still reports all hits inside as 100% Defaults to true for circular zones, always false for quad-based zones.
details	When set to true, reports of an impact include the details of weapon type, pilot name, airframe type, total travel distance, impact velocity Defaults to true
reporter	When set to true, impacts are reported. Defaults to true
reportName	If true, the target zone's name is reported with "INSIDE target area' Defaults to false (zone name not added)
smokeHits	If set to true, impact locations inside a target area are marked with smoke Defaults to false (not marked with smoke)
smokeColor	Color of the smoke that marks hits inside the zone. Possible values for the smoke effect are:

Name	Description
	• “green” or “0” • “red” or “1” • “white” or “2” • “orange” or “3” • “blue” or “4” Only relevant if you set smokeHits to true. Defaults to “blue”
flagHits	Similar to ‘smokeHits’, except that an object is placed on the impact point instead of smoke. Called so because the default object that is placed on the impact location is a flag. Defaults to false (no objects placed on impact point).
flagType	Type Name of the object that is to be placed on the impact location. See here https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB/Statics for a collection of possible objects. Only relevant if you set flagHits to true. Defaults to “Red_Flag”
clipDist	Maximum distance from center of the trigger zone that a hit is reported. Defaults to 2000 (meters = 2km = 1.2 miles)
method	DML method to bang! output flags. Defaults to “inc”
hit!	DML output. List of flags to beng! when a weapon fulfils the hit condition Defaults to <none>
markBoundary	If set to true, the outline of the target zone is marked with a number of objects for visual cues for pilots. All objects belong to Neutral Defaults to false (no marks)
markType	Type Name of the object that is used along the zone’s boundary. See here https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB/Statics for a collection of possible objects. Defaults to “Black_Tyre_RF”
markNum	Number of repeats for the boundary marker per quarter. The number of objects placed is one higher per quarter, so the total numbers of objects placed is (1+markNum) * 4. If, for example, you set markNum to 5, a total number of (1 + 5) * 4 = 24 objects are places around the target zone’s outline. Defaults to 3 (a total of (1 + 3) * 4 = 16 objects along the outline)
markCenter	If set to true, the center of the bomb zone is marked by an object. The object belongs to Neutral. Default to false (no object to mark center)
centerType	Type Name of the object that marks the center. See here https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB/Statics for a collection of possible

Name	Description
	objects. Defaults to “house2arm”
markOnMap	If set to true, the bombZone is marked on the F-10 map. Default to false (no mark in F-10 Map)
mapColor	Four numbers, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the zone’s outline on the map. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to “0.8, 0.8, 0,8 1.0” or #CCCCCCFF – a light gray, fully opaque
mapFillColor	Four numbers, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the zone on the map. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to “0.8, 0.8, 0,8 0.2” or #CCCCCC33 – a light gray, 80% transparent

5.4.20.5 Using the module

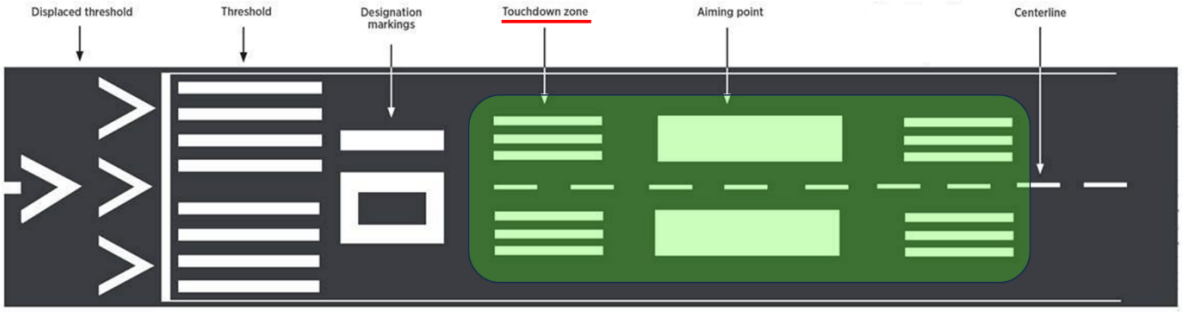
Copy the bombRange source into a DOSCRIPT action that runs at the start of the mission.

Then add at least one triggerZone with a “bombRange” attribute, and add other attributes as required.

5.4.21 TDZ (Touch-Down Zone)

5.4.21.1 Description

Similar to Bomb Range, TDZ is primarily a drop-in training tool, designed for mission creators to add capabilities for training a player's aircraft landing abilities. Since landing is an essential skill, and especially when preparing for carrier approaches, being able to reliably put an aircraft down on a precise point of the runway is a necessity, TDZ can provide mission designers with a convenient and simply way to set up a trainer.



BASIC FUNCTIONALITY

TDZ observes players, and when they touch down inside one of its 'touch down zones', it tracks landing details and – after they come to a full stop – sums up the landing so they can use this data to improve their landings

- Grading initial touch-down point inside the touch-down zone
- Touch-down
- Total run (distance) until full stop
- Deviation from centerline
- Heading on touch down and deviation from runway heading
- Deceleration time (initial touch-down until stand-still)
- Number of hops/bounces

If the aircraft leaves the runway before coming to a full stop, the landing is disregarded.

SETUP

Since TDZ is designed to work exclusively with runways, it automatically homes in on the closes runway on your map, so set-up is quick, and does not require precise placement – just place the TDZ zone close to the runway that you want to use (note: for airfields that provide more than one runways, you may need to re-position the trigger zone slightly in order to allow TDZ to attach itself to the correct runway.



TDZ can draw its zones onto the F10 in-game map for visualization, and mission designers can use this feature to tweak the TDZ's setup for their training purposes.

By default, a TDZ starts at the runway threshold, and extends 610m (2000 ft) in the runways direction. Using attributes, you can curtail the zone for your specific needs.

LEFT, RIGHT OR BOTH?

All runways have two directions that you can land on, so you use TDZ to define separate TDZ for each direction. By default, TDZ simply mirrors your setup for both directions.

However, many runways in DCS (and the real world) are asymmetrical, and you may want to either only provide a sings TDZ (which makes sense for missions since you can design your mission for one approach direction) or separate TDZ for each landing direction.

TDZ supports this through the attributes “left” and “right”. If you set either attribute’s value to false, that approach direction is disabled, and a touchdown in that direction is ignored. In Terms of TDZ, ”Left” means in the direction of the map’s runway definition, “Right” means the opposite direction.

And now comes the ugly part: by looking at a map, short of experience there is no way of knowing which direction “left” or “right” is going to be, that entirely depends on how a runway is defined by the map’s creator (ED or third party). Each airfield defines their runways as their vendor sees fit, and you will have to experiment yourself to find out which is which. For example, Kobuleti’s Runway 07/25 internally is defined as “Runway 25”, meaning that “left” means landing with a heading of 250°, while “right” for Kobuleti means landing with a heading of 70°

By adding two TDZ trigger zones, one for left and right you can define separate touch-down zones for each direction, but (unless you have done so before and know which is which) must experiment to see what the airfield’s ‘left’ is.

DEFAULT MARKINGS ON THE F10 MAP

TDZ by default draws a black outline of the runway, and green filled zones for each TDZ onto the F10 Map view. TDZ provides built-in fine support for outline and fill colors, so you are not restricted to the defaults. This visualization feature is mostly intended to help mission designers to correctly map out each runway’s TDZ and then disable them by disabling the ‘visible’ attribute, but you can decide to leave visibility of these marks enabled for your players for orientation. Note that these markings are only visible on the F10 Map.

Note that TDZ supports the full RGBA color model for both frame and fill colors, setting any of these color’s alpha value to zero will turn off that feature (an alpha of 0 makes the color fully transparent). Use this feature to, for example, selectively disable the runway frame, but allow the TDZ be displayed.

LIMITATIONS

TDZ only works with the airfields provided by the map. It automatically recognizes all airfields on each map, but it does not work with FARPS or naval units (aircraft carriers etc):

5.4.21.2 Dependencies
TDZ requires dcsCommon and cfxZones

5.4.21.3 Module Configuration
To configure aspects of TDZ via a configuration zone,

- Place a Trigger Zone anywhere on the map
- Name it “tdzConfig” (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 for all TDZ on the map. Default is “false”

5.4.21.4 ME Attributes
Note: the settings are applied to the runway that is closest to the center of the trigger zone.

Name	**Description**
TDZ	Marks this trigger zone as TDZ and use the center of this zone to determine which runway on the map to attach itself to. **MANDATORY**
verbose	When set to true, TDZ reports debugging information for this zone. This is particularly useful to discover the “Left” direction of an airfield’s runways Defaults to false (verbosity off)
helos	When set to true, the Zone also reacts to helicopter landings. Defaults to false
extend	Extend the length of the runway (as reported internally by the map’s DB) by this amount of meters. Note that the extension is added to both ends. Negative numbers shorten the runway on both end by this amount. Defaults to 0 (meters)
expand	Expands the width (i.e. widens) of the runway (as reported internally by the map’s DB) by this amount of meters. Note that the width is added to both sides. Negative numbers make the runway narrower on both sides by this amount. Defaults to 0 (meters)
starts	Offset (in meters) from the runway threshold (as defined by DCS’s airfield DB) where the TDZ starts Defaults to 0 (meters, directly at threshold)
ends	Offset (in meters) from the runway threshold (as defined by DCS’s airfield DB) where the TDZ ends. Defaults to 610 (meters = 2000 ft)

Name	Description
landed!	Output to send a signal when an aircraft has successfully landed (touched down, did not leave the runway, and come to a complete stop on the runway (if 'opposite' is true – as per default – the direction of the landing is ignored) Defaults to <none>
touchdown!	Output to send a signal when an aircraft has touched down on the runway Defaults to <none>
failed!	Output to send a signal when an aircraft has touched down, and subsequently left the runway before coming to a complete stop (for whatever reasons, be they intentionally or an unscheduled rapid disassembly) Defaults to <none>
rwFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the runway. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to "0.0, 0.0, 0.0, 0.0" or "#00000000" transparent black
rwFrame	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the runway's outline. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to "0.0, 0.0, 0.0, 1.0" or "#000000FF" opaque black
tdzFill	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to fill the TDZ. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to "0.0, 1.0, 0.0, 0.25" or "#00FF0040" transparent green
tdzFrame	Four numeric values, separated by comma (e.g., 1.0, 0, 0, 1.0) that define the RGBA (red, green, blue, alpha = opacity) values for the color used to draw the runway's outline. RGBA values each range from 0.0 to 1.0 Alternatively, you can use the #RRGGBBAA format, where RR, GG, BB and AA are each hexadecimals (as is used in HTML/CSS to denote colors) Defaults to "0.0, 1.0, 0.0, 1.0" or "#00FF00FF" opaque black
visible	When set to true, TDZ draws a frame and fills the runway with the selected colors and then draws the TDZ(s) on top.

Name	Description
	Defaults to true (draw runway and TDZ)
left	When set to true, the ‘left’ direction (runway’s “main” landing direction, as reported by the map’s DB) of this runway is active, landings in this direction are reported, and the TDZ in this direction is active. Defaults to true
right	When set to true, the ‘right direction (opposite to the airfield’s reported ‘main’ landing direction) of this runway is active, landings in this direction are reported, and the TDZ in this direction is active. Defaults to true

5.4.21.5 Using the module

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

Place a triggerZone close to the runway that you imbue with TDZ abilities, and add A TDZ attribute; add other attributes as required.

5.4.22 Scribe (player logging)

5.4.22.1 Description

Scribe is a module that keeps records of player's actions. It's most useful when used on server missions, and since it supports persistence, it encourages players to return and play the mission more often and increase their stats.

Scribe is a base record-keeping module that other DML modules can (and will) use to track player achievements. Players can at any time query scribe for their own statistics – and (for reasons of privacy) only their own statistics.

Scribe currently tracks, per airframe

- Time in airframe
- Start-ups (hot-starts do not count)
- Landings
- Take-offs
- Crashes

When used with other modules, it also gains the ability to track:

- Rescues (csarManager)

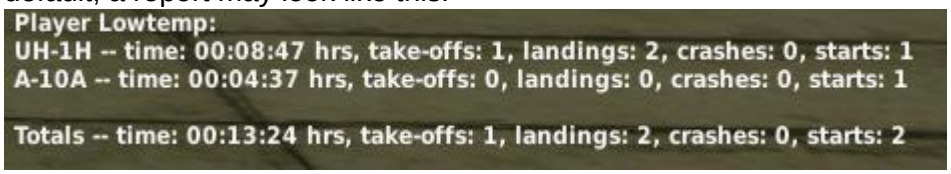
If you want to use scribe to simply gather information on your server's usage, you can disable scribe's built-in Communications → Other UI.

Privacy Advisory

Scribe does not collect any data on players that isn't also available from logs, and it will not allow other players to view any data except their own. Note that the server owner can access all data collected by scribe, which, again, doesn't contain any data that wouldn't be available by trawling the logs. Using scribe will not endanger or compromise your, or your player's privacy. No player-identifying data except for their on-line name is retained.

A Scribe's Report

Scribe can report to a player the data it has collected in a flexible, customizable form. By default, a report may look like this:



A report always lists details by aircraft type (here a UH-1H Huey and an A-10A Hog) and then totals. Per line, scribe can display

- Time in aircraft type
- Take-offs
- Landings
- Crashes
- Aircraft Start-ups

- Rescues (requires csarManager)

You can use scribe's config zone to change what information is shown, and what they are labeled with:



Above uses custom labels for all data, and it adds rescues (which is only available to helicopters) data, with a custom "RSQ" label. Use the *lxxx* (e.g. *lLandings*) attribute to control a datum's label, and *xxx* (example *landings*) itself to control if scribe should report the datum itself.

Sharing data with other missions
Your mission can share scribe's data with other missions provided that

- You have enabled persistence
- You have added the 'sharedData' attribute in the config zone and supplied a name under which this mission's scribe data is shared.

5.4.22.2 Dependencies
Scribe requires dcsCommon, cfxZones and cfxMX

Optional: radioMenu if you want to take place scribe's UI in your own menu.

Scribe requires persistence to persist data

Scribe records all successful rescues when csarManager is active.

5.4.22.3 Module Configuration
To configure aspects of scribe via a configuration zone,

- Place a Trigger Zone anywhere on the map
- Name it "scribeConfig" (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"
hasGUI	If set to true, players can query their own data from the Communications→Other menu. Defaults to true (UI enabled)
uiMenu	Name of the Menu to display in the UI if it is enabled (see above). Defaults to "Mission Logbook"
greetPlayer	If set to true, players are shown their current statistics whenever they enter a plane Defaults to true (players are greeted)

Name	Description
byePlayer	When set to true, whenever a player leaves an aircraft, their current total time in that type are broadcast to all connected players. Defaults to true (broadcast total time in type for player on leaving aircraft
landings	If true, scribe will report landing stats (note: scribe record all data. This attribute merely controls if this data is *displayed*) Defaults to true
lLandings	Label to use that precedes landing data. Defaults to “landings:“
departures	If true, scribe will report take-off stats (note: scribe records all data. This attribute merely controls if this data is *displayed*) Defaults to true
lDepartures	Label to use that precedes take-off data. Defaults to “take-offs:“
startups	If true, scribe will report engine start-up stats. Only applies to aircraft that start cold. Start-ups from aircraft that start hot are discarded even if they are shut down and then re-started. (note: scribe will records all data. This attribute merely controls if this data is *displayed*) Defaults to true
lStartups	Label to use that precedes engine start-up data. Defaults to “starts:“
crashes	If true, scribe will report crash stats. Ejecting and being KIA also count as crashes, but successful ditchings (without loss of pilot life) do not count as a crash (note: scribe record all data. This attribute merely controls if this data is *displayed*) Defaults to true
lCrashes	Label to use that precedes crash data. Defaults to “crashes:“
rescues	If true, scribe will report the number of successful rescues, as reported by the csarManager module (note: scribe records all data. This attribute merely controls if this data is *displayed*) Defaults to true
lRescues	Label to use that precedes rescue data. Defaults to “rescues:“
landingCD	Number of seconds that have to occur between landings for a second landing to count as a landing. This is used internally to differentiate a ‘hop’ from a landing. Defaults to 60 (seconds)
sharedData	When present, this defines the name under which scribe shares its data via persistence. Any other mission that uses scribe and has defined the same name, will read from and write to scribe’s *shared* data file with that name. **Example**: The missions *“Angels of Caucasus”, “Abgels of the Levant – Sinai”* and *“Angels of the Levant - Syria”* all use “Angels” as value to their respective “sharedData” attributes in scribe’s config zone. Now, whenever the mission loads or saves data (via persistence), instead of using the mission-specific data, it reads and writes the shared data file “Angels.txt” that is located inside the shared data folder (usually “DML-Shared-Data”) Defaults to: <none> (data is not shared with other missions)

Name	Description
attachTo:	When present, scribes UI attaches to the trigger zone named as value. This trigger zone must have a “radioMainMenu” attribute. Note that if you use this attribute, you must also add the radioMenus module to your mission and that module **must** load before scribe. Defaults to <none> (UI installs at F10 Other level)
tickTime	Tracking resolution (in seconds) that scribe uses to track player units. This limits possible measuring errors from pauses and/or reloads. Defaults to 5 (seconds)

5.4.22.4 ME Attributes

Scribe currently uses no map-specific zone, all features are controlled through the config zone

5.4.22.5 Persistence

Scribe automatically supports persistence.

5.4.22.6 Using the module

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

Optionally, add a config zone and curtail scribe’s behavior.

5.4.23 Civ Helo

5.4.23.1 Description

This module provides civilian (non-aggressive) helicopter traffic to your map. Similar to civAir, Helicopters shuttle between civHelo zones, and you can control both which types of helicopters are used, and which liveries they use.



Flights

CivHelo creates flights of helicopters between departure and arrival points on the map. Mission designers can create these points by placing trigger zones and add the relevant attributes.

A flight is always a single helicopter, and the type of that helicopter is chosen at random from a set of types that you can curtail.

By default, a flight is chose from the following set:
CH-47D, CH-53E, Ka-27, Mi-24V, Mi-26, Mi-28N, Mi-8MT, OH-58D, SA342L, SH-60B, UH-1H, UH-60A

Each aircraft also randomly chooses a livery from a known set. You can control both types and liveries

Departure and Arrival Zones

CivHelos start (appear) in departure zones, and travel to, land, and disappear in arrival zones. You control if a zone can be a departure zone (where helos start) or arrival zone with the ‘start’ and ‘land’ attributes of that zone. A zone can be both a departure and arrival zone.

When civHelo creates a flight, it ensures that there are both departure and arrival zones available for the flight, that they are sufficiently far apart (→ minDist) and not too far away (→ maxDist). Departure and arrival zones are then reserved for the flight.

Once the flight has sufficiently departed from the departure zone, civHelo frees the departure zone for other civHelo traffic. The arrival zone is only freed after the flight has successfully landed, or the flight was destroyed.

Note that civHelo’s minDist and maxDist allow the following common design patterns:

- **Departure/Arrival “hub”:**
  You can place multiple civHelo zones close together to allow civHelo to start or arrive

multiple flight at these ‘clusters’. If you bunch these zones together and make sure that they are all within minDist, civHelo will not create flights between these points.

- **CivHelo “bubbles”**
  Two civHelo zones that are further apart than maxDist will never be connected by a flight between them. You can use this to make a ‘local’ civHelo hub in one region of the map (e.g. Batumi in Caucasus), and another one at a location sufficiently far away (e.g. Tbilisi)– all you have to do is to ensure that no civHelo zone from either location comes within maxDist of the other location.

A word on arrival and departure locations
Unlike Mission editor, civHelo allows you to place arrival and departure zones in some unusual locations like gas platforms, mountain tops, overgrown or steep inclines etc. Sometimes, using these locations can pay off greatly. Usually, they don’t. Sometimes, adding a single helipad can help. Sometimes, things simply wont work out with current DCS AI:



While it’s no problem to have a civHelo flight originate from an offshore gas or oil platform, DCS’s AI can’t reliably land a helicopter there. Spawning or landing on steep inclines is going to end in disaster more often than not, etc. Make sure that you test your departure and arrival locations before you publish the mission. That being said, a crashed civHelo helicopter does create any problems for the module, it merely proceeds to create a new flight at the next convenient point in time.

Maximum Traffic
At any time, two factors control the amount of civHelo flights that are in the air at the same time:

- The number of free departure and arrival zones. Each flight requires one free departure zone, and one free arrival zone that is within maxDist and at least minDist away. After take-off, a flight relinquishes the departure zone, but the arrival zone remains locked until the flight lands there or is destroyed. So if you are wondering why there are so few flights in the air, check the number of civHelo zones on the map.

- The global maxFlight puts a hard cap on the number of flights that can be created, provided enough free civHelo zones are available. This is to allow mission designers to limit the amount of performance that may be allocated to civilian helicopter traffic AI (which is not a great amount, but being able to limit the impact seems like a good idea)

5.4.23.2 Dependencies
civHelo requires dcsCommon and cfxZones

5.4.23.3 Module Configuration
To configure aspects of scribe via a configuration zone,

- Place a Trigger Zone anywhere on the map
- Name it “civHeloConfig” (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”
country	The country ID to whom all helicopters belong. Defaults to 82 (Combined UN peace force)
ups	Updates per second. Defaults to 1/10 (1 Update every 10 seconds)
maxFlights	Maximum number of helicopters simultaneously under the control civHelo. Defaults to 5
landingDuration	Time between the point when a civHelo-controlled helicopter initiates a landing, and when it is removed from the map. Defaults to 180 (seconds = 3 minutes)
alt	Altitude in meters (AGL) that flights use to fly between departure and arrival points. Defaults to 100 (300 ft)
speed	Speed (in m/s) that flights use to fly between departure and arrival points. Defaults to 30 (m/s = 108 km/h = 60 knots)
maxDist	When picking destination zones for a flight, that destination must not be further away than this distance (in meters) Defaults to 60000 m = 60 km = 37 miles
minDist	When picking destination zones for a flight, that destination must not be closer than this distance (in meters) Defaults to 1000 m = 1 km = 0.6 miles
types	DCS types that civHelo can choose from for its flights. Types are separated by comma (“,”) Defaults to CH-47D, CH-53E, Ka-27, Mi-24V, Mi-26, Mi-28N, Mi-8MT, OH-58D, SA342L, SH-60B, UH-1H, UH-60A

5.4.23.4 ME Attributes

You place civHelo zones on the map to define where helo flights can originate from and where they fly to. Note that with civHeloConfig you can control how far apart these zones have to be at a minimum, and how far they may be apart at maximum, so you can build multiple 'civHelo hubs' on your map.

Name	Description
civHelo	Tells DML that this zone is a civHelo zone that can be used for flights **Mandatory**
land	If set to true, civHelo flights can land here (use the zone as a destination for a flight) Defaults to TRUE (civHelo flight can use this as destination)
start	If set to true, civHelo flights can take off from here (use the zone as a starting point for a flight) Defaults to TRUE (civHelo flights can use this a departure point for a flight)
hot	If set to true, civHelo flights that depart from here start hot. Otherwise, the flight is cold-started Defaults to FALSE (flight is cold-started)
types	The DCS types of helicopters, separated by comma (",") that can start from here. Example: `Mi-8MT, SA342L, UH-1H` Defaults to the same types that are supplied with the config zone.

5.4.23.5 Using the module

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

Optionally, add a config zone and curtail civHelo's behavior.

Add at least two trigger zones with a "civHelo" attribute that can serve as source and destination for flights. Make sure that those two zones fulfil both minDist and maxDist constraints.

5.4.24 Slotty

5.4.24.1 Introduction

The excellent “Simple Slot Block” script developed by Ciribob has allowed mission designers since 2017 to create missions with player slots that can be blocked. This allows, for example, missions with airfields that can be captured and player aircraft on these airfields to only become available to players when their side has captured the airfield/FARP.

DML’s *ssbClient* module implements the “SSB protocol” to allow mission designers just that: aircraft slots being tied to conquerable airfields/FARPs. Other modules in DML use the same “SSB Protocol” to deny player access to aircraft after they crash, and there are many other fun uses for using SSB in your missions to control player’s slot control. Until recently, though, this meant two things:

- The server that runs the mission must have Ciribob’s “SSB” server script installed. Installation is simple, and third-party hosting providers install SSB by default because it is so popular.
- The mission itself must be run in multi-player for the server SSB script to take effect. In single-player SSB does not work for the simple reason that the server isn’t active.

The ‘slotty’ module makes above points moot. It implements the SSB protocol for single-player and multiplayer on the client side, and allows slot-blocking for single and multiplayer. Even better, it chips in in there is no SSB installed on the server and provides a fallback mechanism so you can now create missions that rely on SSB without having to bother if SSB was installed.

5.4.24.2 Description

Slotty provides **single-player slot blocking** and a fallback for **multiplayer slot-blockin**g if the server that is running your mission does not have SSB installed.

Slotty conforms to the **SSB (“Simply slot block”) protocol** and kicks players that are trying to ‘slot’ into an aircraft that is marked as not available. Modules like *ssbClient*, for example, use this protocol to prevent players from slotting into aircraft that are located on airfields or FARPs that belong to the enemy.

If the server running your mission hasn’t SSB activated/installed, or the mission is run as single-player, the SSB module can’t prevent players from slotting into those aircraft. This is where slotty comes in. If you add the module to your mission, that mission automatically gains smart slot blocking for single player, and a fallback for multi-player should SSB not be installed. Note that the player experience is better if SSB is running for multiplayer. In all other instances, slotty will jump in and pick up the slack.

Slotty is a drop-in-and-forget module that does not require any configuration.

5.4.24.3 Dependencies

None. Slotty doesn’t even require DML

5.4.24.4 Module Configuration
None. Slotty works out of the box

5.4.24.5 ME Attributes
None.

5.4.24.6 Interactive enable/disable
You can disable slotty from blocking slots by setting the value of the flag “**noSlotty**” to any value greater than 0 (Zero)

5.4.24.7 Using the module
Add Slotty to your mission during AT START as a DOSCRIPT. Smile.

5.4.25 Reaper

5.4.25.1 Description

The reaper module allows you to quickly place a drone in the air over a location and have it automatically lase enemy ground units. All that it requires is placing a trigger zone and add an attribute.

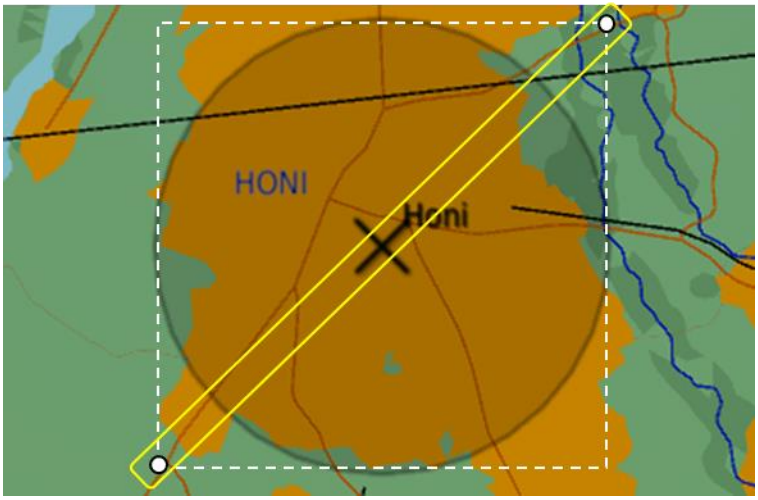


That being said, reaper is a fully fledged DML module and comes with all the bells and whistles that you'd expect, including a UI.

Reaper or Predator

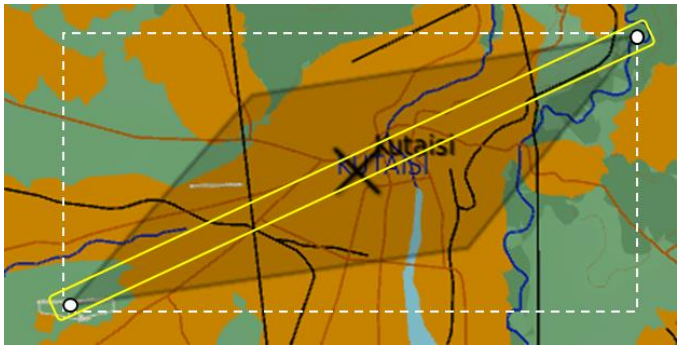
There currently are two different drone types that you can choose from. By default, the reaper module uses an MQ-9 Reaper. Alternatively, you can choose an RQ-1A Predator type drone. Which type of drone you choose has no impact on lasing. A Reaper has a higher service ceiling and moves a bit faster than a Predator drone. Neither is equipped for combat; both merely provide lasing support (and battlefield intelligence if the map settings are set to 'fog of war').

Operational area



With placing the trigger zone on the map, you also define the area of operation and flight path of the drone. When you place a circular trigger zone, the drone will fly a racetrack pattern between the two points that are defined by the 'bottom left' (southern-western) and 'upper right' (northern-eastern) points that form the corners of the square that completely encompasses the trigger zone circle. This means that, statistically, the drone will spend a large fraction of the time over the center of the target zone. Due to the nature of drones and flight, the drones will extend their flight path slightly over the two points; if you choose the target zone's size to be too small, the drone will extend to the minimal size possible to maintain a good pattern.

Similarly, if you define the area of operations as a quad-based trigger zone, the drone will create a racetrack pattern from the ‘bottom left’ corner of the smallest surrounding rectangle to the ‘top right’ corner, again making the drone always cross over the center of the zone.



Note that once a drone has detected a target and starts tracking (lasing) it, it will alter its flight path to continue to track the target.

Note also that drones are sitting ducks when it comes to enemy fire, be it SAM or aircraft, they do not defend themselves (nor do they have the means to).

Auto-Lasing
Drones automatically scan for enemy units, and once they find a target, will automatically lase the target. Unless specified otherwise, the laser code is 1688. Once a target is found, the drone continues to lase the target until it is destroyed.

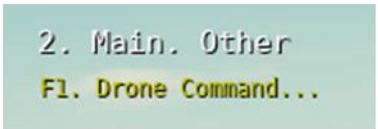
Target Cycling
Once a drone is engaging a group, players can tell the drone to cycle the target. Cycling will cause the drone to choose the next target in the group (or first target in the group after the last). Note that all cycled targets are within the same group, and the drone will only look for other groups once all targets have been destroyed, or have otherwise disappeared (e.g. hidden from view).

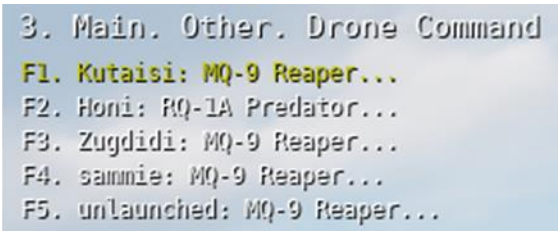
Visual Range
A drone’s visual range is determined by its altitude. As a rule of thumb, the range is equal to it’s altitude: if a drone is at 5000m (15’000 ft) altitude, it can detect targets up to 5000m (15’000 feet away). Note that terrain can mask a unit’s visibility. This means that even though a unit may be within detection range, a drone still may fail to detect it if it’s masked by terrain.

As soon as a drone detects and locks on to a target, it sends a message to all players from the same side with information about the target’s location and laser code:

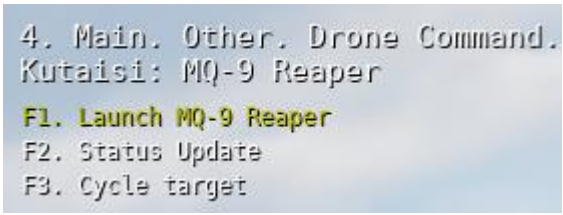


Reaper’s built-in “Drone Command” UI (Individual Drones)
Reaper comes with a built-in UI (that you can disable or move to other locations) to manage all drones. Using this UI you can give players access to all drones that reaper manages





Players select the drone (zone) that interests them, and then receive a drone-individual command menu:



Note that since DCS is limited to 10 menu items per menu, Reaper's built-in UI supports at most 10 Reaper zones per side. If you want to use more, you'll need to build your own reaper control menus via the radioMenu module (see the 'secret drone' option in the demo)

Per drone, you have three options:

- *Launch the drone*
  If the drone is already launched, nothing happens. If cost is activated and the bank module has loaded, funds are checked and the drone is only launched if there are sufficient funds available. After launch, the funds are withdrawn from that side.

- *Status Update*
  The drone sends a report about the target it is currently lasing, and the entire group's composition:
  

- *Cycle Target*
  This causes the drone to step through all targets in the group. Players can use this to prioritize targets in the group. In above example, cycling will cause the drone to target the T-90 next, then the BTR-80, then back to the AAV7. The drone cannot cycle to other groups.

  NOTE:
  Some DCS aircraft (for example the Kiowa) simulate a video feed from the drone. Since this is only simulated, and there currently is no method to determine which unit a drone targets, cycling the target will not actually change targets in the video feed, not will the displayed unit be guaranteed to be the target that the drone is currently targeting. When DCS provides an API to interface with drones, this function may be updated.

Reaper's UI supports radioMenu's radioMainMenu layer via the attachTo: attribute, so you can move the reaper menu from the "Other" level to others that better suit your mission.

Drone Summary (Status)
Reaper can provide a summary status of all drones, split into three groups:

- Tracking/lasing (with target type and location of target, plus laser code)
- Loitering (with drone name and type)
- Drones that are ready to launch (with required funds if that is enabled)

Note that all drones from reaper zones that have their statusUI attribute set to false will not show up in this report. Here's an example:



DML triggering
All reaper zones provide inputs to selectively trigger a drone launch, drone status message, or target cycling.

Additionally, reaper also supports triggering a fully drone status report (all drones) per coalition.

Integration with other modules
Reaper provides automatic integration with the following modules:

- **radioMenu**
  Reaper supports an "attachTo:" attribute to relocate reaper's "Drone Command" menu to a radioMainMenu zone of your choice

- **bank**
  When reaper's cost ability is enabled and the bank module is present, reaper only launches a drone when the there is enough funds in the bank for the faction, and automatically deducts the amount.

5.4.25.2 Dependencies
reaper requires dcsCommon and cfxZones

Optionally requires radioMenus (for attachtTo:) or bank (for priced launches)

5.4.25.3 Module Configuration
To configure reaper via a configuration zone,

- Place a Trigger Zone anywhere on the map
- Name it “reaperConfig” (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”
actionSound	Name of the sound file to play when a communication is made (e.g. a drone is launched or starts tracking a unit) Defaults to <none>
UI	When set to false, reaper does not install its own UI for getting drone status reports or launching drones. Defaults to true (reaper installs its own UI)
menuName	Name of the reaper menu (when active) Defaults to “Drone Command”
useCost	Use reaper’s drone cost ability. Requires that the bank module is also present in the miz. When active, a faction can only launch drones if they have sufficient funds. Drone cost is set individually per reaper zone. Defaults to
blueStatus?	Input to trigger a status report of all drones for the BLUE side (only visible to BLUE players). Lists only those drones that have not set their statusUI attribute to false. Defaults to <none>
redStatus?	Input to trigger a status report of all drones for the RED side (only visible to RED players). Lists only those drones that have not set their statusUI attribute to false. Defaults to <none>
attachTo:	If present, the reaper UI attaches to the trigger zone that is named as value. That trigger zone must have a *radioMainMenu* defined, and radioMenu must run before reaper.

5.4.25.4 ME Attributes

You place reaper zones on the map to define where the drone is going to patrol. The patrol route is defined as the ‘cross section’ of the zone (meaning the size of the zone is relevant). The reaper zone defines all drone-individual properties.

Name	Description
reaper	Tells DML that this zone is a reaper zone that is patrolled by a drone. The value of this attribute defines the type of drone. Possible values are • Reaper – the drone is a type “MQ-9 Reaper”, max altitude 9500m (28000ft), speed 200 km/h (110kts) • Predator – the drone is a type “RQ-1A Predator”, max altitude 7500m (22000ft), speed 120 km/h (65 kts)

Name	Description
	Defaults to “Reaper” **Mandatory**
alt	Altitude (in meters) for the drone’s racetrack pattern. Note that the altitude also defines the drone’s visual range, which is roughly equal to the altitude. Predator and Reapers have different ceiling altitudes.
coalition	Faction that owns the drone. Possible values are • 1 or “red” • 2 or “blue” • Note that neutral drones are automatically switched to BLUE Defaults to BLUE
onStart	If set to true, the drone is launched on mission start, at no cost. Defaults to true (drone launches at mission start)
code	Laser code for lasing. Defaults to 1688
doSmoke	Visual aid for targeting. When a drone starts targeting a unit, it can also place smoke at the location of the target. The smoke marker lasts circa 5 minutes and does not move with the unit. Defaults to false (no smoke when marking units)
smokeColor	Color of smoke to place at the location of the target. Valid values are • 0 or “green” • 1 or “red” • 2 or “white” • 3 of “orange” • 4 or “blue” Defaults to “blue”
cost	Only used when config’s ‘useCost’ is enabled and the ‘bank’ module is loaded. Cost in ‘fund units’ to launch a drone. Does not affect initial launches if the onStart attribute is true When actice, the cost is included in reaper’s launch UI Defaults to 700
autoRespawn	If a drone is disappeared (for any reason), it is instantly replaced with a new one (for free) if this attribute is set to true Defaults to false
launchUI	If set to true, this reaper zone is included with the reaper UI Defaults to true (zone is included with launch UI)
statusUI	If set to true, this zone’s drone is included in the global status reports. Defaults to true (drone is included in reaper status reports)
launch?	Input that when triggered launches this zone’s drone. When cost enabled and the bank module present, a funds check is performed and the drone is only launched if sufficient funds are present. Defaults to <none>

Name	Description
status?	Input that when triggered outputs this zone's drone status to the owning side. Defaults to <none>

5.4.25.5 Using the module

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

Optionally, add a config zone and curtail reaper's behavior.

Add at least one trigger zone and add the 'reaper' attribute.

5.4.26 Inferno

5.4.26.1 Description

This module can create randomized, growing fires inside a zone that can be put out in conjunction with the *airtank* module. Inferno creates 'fire cells' inside a trigger zone that can spread to other cells inside the zone (but not cross-zone), and can be used to create impressive dynamic scenery (for example after an aircraft crashes in a forest) and provide tasks for aerial firefighters.



Note that currently in DCS, all fires are visual effects only: they do not affect (outside of visibility) other units nor the local weather. There are no updrafts or air currents, the flames (and smoke) are not affected by rotors, and likewise, flying through or inside a fire effect does *not* adversely affect units.

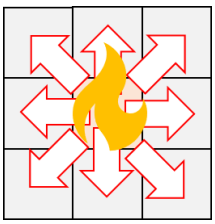
INFERNO ABILITIES

Area of effect

Mission authors use a trigger zone (circular or quad) to define the area in which the fire can spread. It will never spread outside the zone, and the module provides logic to prevent fire cells to sprout on rivers, lakes, roads, or deep water.

Start & Spread

Inferno models the conflagration inside the trigger zone as a collection of *fire cells*. How many cells inhabit a zone is controlled by the *cellSize* attribute in conjunction with the surface area of the trigger zone. All infernos start from a single fire cell that starts small, and then grows if there are potential fire cells located nearby.



To start the fire inside a zone, send a signal on the *ignite?* input. The first fire cell is either picked by random (if *rndLoc* is true) or near the center of the zone. Once a fire has grown sufficiently, it can spread to neighboring cells, provided that the cell is still inside the trigger zone and eligible to burn: it has

- has not been doused (via *airtank*),
- is not inside a body of water or on a road,
- has enough fuel and therefore has not burned out
- the number of burning cells in this zone is below the maximum

To model a fire cell, Inferno uses DCS's 'Fire FX', a purely visual effect that is a 2D 'billboard': a virtual surface that always is oriented towards the viewer, with pixel art (flames and smoke) drawn onto the billboard. For the biggest (DCS-provided) flame effects, these billboards are some 100m wide, and therefore a fire cell defaults to side length of 100m. Since fires in DCS are not volumetric, getting close to them is currently an underwhelming experience. Hopefully, later DCS releases change that.

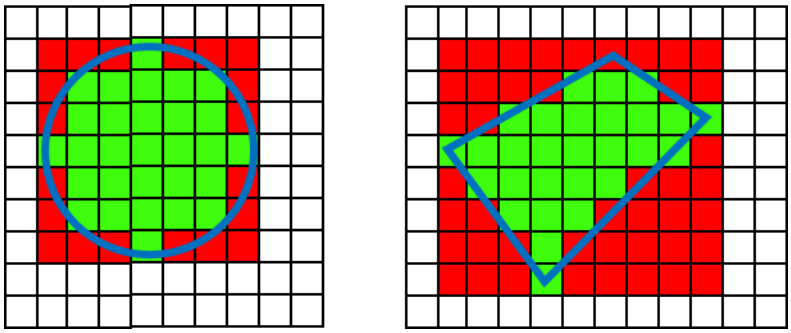
Limiting spread of fire

When you place a trigger zone and designate it as an inferno, the module defaults to a cell size of 100m x 100m. Therefore, if your trigger zone is 2km x 3km in size, it can contain up to 20x30 = 600 fire cells. If you place no restrictions on that zone, and set a match to the zone (via ignite?), after a while it will be fully ablaze, with 600 flame and smoke effects. Although possibly impressive, it can be quite taxing to any GPU. The *maxSpread* attribute caps the number of times that a fire can spread to provide more versatility and greater random, inferno provides mission designers with the ability to restrict the number of fire cells that a fire can spread to. This allows you to place an inferno over a large area, yet allow a fire only to grow to a specific, smaller size for randomization and better performance. For example, an inferno with a maxSpread of 7 can at most have 8 active fire cells inside: one initial cell, plus 7 cells that it can spread to. If there are fewer cells, or the shape or ground does not allow the fire to spread further, that inferno can't reach its maximum size.

Fire cells spread from the center of one cell to the surrounding eight cells, using a randomized propagation algorithm. A fire can only spread when enough criteria are met. Note that currently, the direction of the prevailing winds are not considered.

Zone Shape vs fire cells

It is not always simple to correctly foresee how many fire cells fit into a trigger zone. All fire cells are square, and for simplicity, DML first creates a rectangular axis-aligned bounding box (AABB) around the trigger zone, and then partitions that rectangular area into squares of *cellSize*. Only those squares whose center falls inside the trigger zone are regarded as fire cells:



In the example above, only the green squares are considered fire cells, while the red squares are discarded and will not be used to house a fire cell

Iteration Speed

Inferno simulates the spread (and dousing) of fires based on an internal clock called *fireTick*. Each time it calculates the size of the fire for the next cycle. By default, each cycle lasts 10

seconds, so inferno calculates 6 cycles per minute. This allows fires to spread quickly (within minutes) and respond quickly after being doused with water. If you plan to use inferno purely for visual effects, I recommend that you increase iteration speed, as you do not need the (small) overhead to calculate the next cycle.

If you plan to use inferno with the *airtank* module to provide fire-fighting abilities, you may want to think about tweaking the *fireTick* value to adapt the response of fires to reality. Setting it to 1 makes the fires spread and respond very fast, while setting it to larger values like 20 makes the overall response more realistic but perhaps less fun.

Fuel
A fire cell in inferno consumes fuel while burning. When all fuel is consumed, that cell dies out. This is used mainly in conjunction with the *airtank* module, as the flame retardant that the aircraft drop removes fuel from the fire cell. If you increase the amount of fuel per fire cell, it takes more retardant per cell to douse the fire.

Fire zones can be set to *eternal* – if enabled (default), a fire cell burns without consuming fuel until an airtank delivers some retardant into the cell. From that moment on, the cell consumes the remaining fuel. The time it takes for a fire cell to burn out is tied to the fireTick attribute. A fire cell burns an amount of fuel proportional to the flame size per tick, so if your tick time is longer, it burns longer with the same amount of fuel.

Extinguished Cells / Extinguished Inferno / dousing
Fire cells extinguish once they have burnt all fuel. Once a fire cell is extinguished, it can't reignite. Once all fire cells inside the inferno zone are extinguished, the zone can send a signal via the *extinguished!* output.
You can also douse an inferno by sending it a signal on the *douse?* Input, which will extinguish all fire cells that are currently burning.

Once the entire inferno has been extinguished, it can be re-ignited with the *ignite?* Input. All fire cells are reset and the fire is newly randomized if that option is enabled.

5.4.26.2 Dependencies
Inferno requires *dcsCommon* and *cfxZones*.

5.4.26.3 Module Configuration
To configure inferno via a configuration zone,

- Place a Trigger Zone anywhere on the map
- Name it "infernoConfig" (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"

Name	Description
fireTick	Time (in seconds) per 'burn cycle'. The larger the number, the longer a fire burns. Defaults to 10 (seconds)
cellSize	Size (in meters) per side for the fire cells inside the zone. All fire cells are square. This value can be overridden individually by inferno zones. Defaults to 100 (meters, roughly the size of a maximum size fire effect in DCS)
fire#	Direct output that always contains the number of inferno zones that have at least one burning fire cell inside them. Defaults to <none>

5.4.26.4 ME Attributes

You can place inferno zones anywhere on the map except over deep water (well, you can, but it won't burn). The zone defines the area in which fire cells are placed, and the number of cells inside the zone is determined by the zones size, shape and the *cellSize* module config attribute.

Name	Description
inferno	Tells DML that this zone is an inferno zone. The value of this attribute is ignored. **Mandatory**
cellSize	Size (in meters) per side for the fire cells inside the zone. All fire cells are square. This value overrides the value given in the config zone. Defaults to the config value for cellSize.
fuel	Amount of fuel for each fire cell inside the zone. Once a cell ignites, it spends fuel while burning (except when *eternal* is enabled, see below). When all fuel is spent, the fire cell extinguishes and can't re-ignite until the entire inferno zone resets Defaults to 100
rndLoc	When set to true, the location of the first fire cell to ignite is randomized. This can randomize shape and size of the resulting conflagration. Defaults to true (randomized fire starting cell)
freeBorder	When set to true, the fire cells inside the zone are surrounded by a 'ring' of 'dud' (un-ignitable). Use this to visually better align an inferno with geologic peculiarities of the map. Defaults to true (add an inside border)
eternal	If set to true, fires inside a fire cell do not consume fuel until they receive a signal from an outside source (usually the *airtank* module), making all cells, once ignited, eternal. Defaults to true (eternal burn)
stagger	When set to true, the center of the flame effect is randomized inside the fire cell, making the entire inferno look more natural. Defaults to true (randomized FX location)

Name	Description
canSpread	When set to true, a fire cell can ignite neighboring fire cells. Defaults to true (fire can spread to other cells)
maxSpread	The maximum amount of fire cells that a fire can spread to inside this zone. Defaults to infinite
onStart	If set to true, the inferno in this zone is started when the mission starts up Defaults to false (inferno waits for ignite? input)
ignite?	Input to tell this zone to start a fire by igniting a single fire cell inside. If the zone has already started a fire, this signal is ignored. Defaults to <none>
douse?	Input to tell the zone to extinguish all actively burning fire cells inside this zone, and reset it to re-start via *ignite?* Defaults to <none>
extinguished!	Output that bangs on the connected flag when all burning fire cells have been extinguished and the inferno cell is no longer burning. Defaults to <none>

5.4.26.5 Using the module

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

Optionally, add a config zone and curtail inferno's behavior.

Place at least one trigger zone and add the 'inferno' attribute.

5.4.27 Airtank

5.4.27.1 Description

Airtank is a drop-in module that facilitates firefighting by players using helicopters.



SUPPORTED UNITS

By default, airtank is configured to provide its fire-fighting abilities to the following DCS modules with retardant capacities (in kg):

- UH-1H Huey (1200 kg)
- Mi-8 MT Hip (4000 kg)
- Mi-24P Hind (2400 kg)
- CH-47 Chinook (9000 kg)
- OH-58 Kiowa (500 kg)

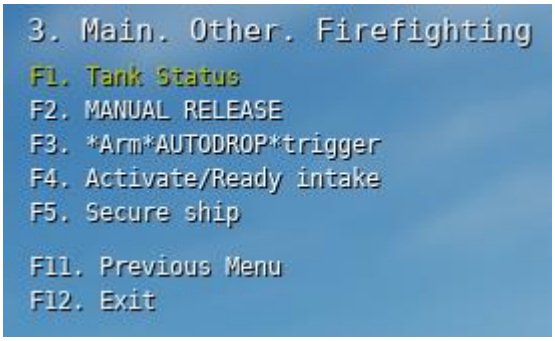
Note that these capacities presume a 50% (i.e. *half*-full) fuel tank; some helicopters will be unable to lift off with full fuel tanks and a full retardant load.

You can change the supported types and capacities (e.g. to add the Blackhawk or remove a supported type, or to change a type's capacity) with the *airtanks* attribute in the config zone.

HOW TO USE

The airtank module allows players to use certain types of aircraft to be used for firefighting. For this, it makes a number of functions available to players:

- Loading flame retardant
- Dropping flame retardant
- Miscellaneous



Loading flame retardant:

There are two ways for players to load retardant/water:

- When inside a refill zone (an airtank zone): Via Communications→Other→Firefighting→Activate/Ready Intake to fill up your flame-retardant tanks. Filling your tanks is instantaneous. (Note that the "Firefighting" menu name, and location on the Other menu are customizable)

- When hovering over open water (lake/river), and at 10m AGL or less: Via Communications→Other→Firefighting→Activate/Ready Intake. Here, the pumps acquire 100 liters (kg) of water per second. For example, filling up the Huey's retardant tank (1200l capacity) takes 12 seconds, while topping off the Hook at that rate takes 90 seconds. Pumps disengage automatically when the tank is full.

Dropping flame retardant
Plaers can drop the contents of their flame-retardant tank in multiple ways. Once started, the process of unloading cannot be stopped, the tanks empty at a throughput of 1000l/second:

- **Manual drop**
  Via Communications→Other→Firefighting→Manual Release. This empties the contents of the helicopter's tanks at 1000kg/sec until they are empty. The impact point is derived from the player's current altitude, speed and direction.
- **Altitude-triggered**
  The player can only initiate this when above 300 m AGL (900 ft). To start the process, they have to choose Communications→Other→Firefighting→Arm Autodrop Trigger. Once the player's altitude drops below 100m AGL (328ft), release is triggered.

Note:
Altitude-triggered release is more difficult, and intended for advanced pilots who can judge their approach.

Once release has started, it ends after the *entire* contents has been emptied. The drop pumps are rated at 1000kg/sec, meaning that the Huey (1200l default capacity) is empty after 1.2 seconds, while the Hook (9000l default capacity) has 9 seconds of water/retardant, and can therefore cover larger areas.

Miscellaneous
Players can use Communications→Other→Fire Fighting→Secure Ship to safe the auto-release trigger and/or intakes.

Putting out which fires?
Airtank is designed to put out fires that are controlled by the inferno module. You cannot use it to douse other fires (e.g. crashed aircraft, cooking off vehicles etc).

Changing Helicopters and their Capacities
The airtank module comes pre-configured to be used with a number of helicopters and their tank capacities. You can define your own set of aircraft and capacities per type by adding a trigger zone and naming it "airTankSpecs", and then add attributes and values, one each per type. This replaces all default types and capacities.

The format is as follows:

Tyle Name	Capacity
[DCS Type Name of aircraft] Example `UH-1H`	A number to define the capacity of this type's flame retardant tank in kg Example `1200`

Note that you can easily exceed the type's lifting capability. If you do, players may not be able a fully filled aircraft of that type.

5.4.27.2 Dependencies
Airtank requires *dcsCommon* and *cfxZones.*

Although not required, airtank makes sense only if you also add the *inferno* module to your mission.

When using "attachTo:", the radioMenu module is also required.

5.4.27.3 Module Configuration
To configure inferno via a configuration zone,

- Place a Trigger Zone anywhere on the map
- Name it "airtankConfig" (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
menuName	Name of the menu installed into the Communications→Other tree for the airtank interface. Defaults to "Firefighting"
actionSound	Name of the sound file to play to accompany an action event Defaults to "none"
blipSound	Name of the sound file to play once a second while the drop trigger (ALT) is armed. Defaults to actionSound
releaseSound	Name of the sound file to play (once per second) when the helicopter releases flame retardant Defaults to actionSound
pumpSound	Name of the sound file to play (once per second) when the intake pumps are engaged.

Name	Description
	Defaults to actionSound
attachTo:	Name of the radioMainMenu zone that airtank's player menu attaches to. Defaults to <none> - attaches to "Other…"

5.4.27.4 ME Attributes

You can place fill stations that can instantly refill a helicopter with flame retardant when they land inside the zone. To create such a zone, place a trigger zone in ME, and add the following attributes:

Name	Description
airtank	Marks this zone as an airtank refill zone The value of this attribute is ignored **Mandatory**
capacity	Maximum amount of Flame retardant that can be stored in this refill zone. Defaults to 99999999 (essentially infinite)
amount	Amount of flame retardant that is stored in this refill station at the start of the mission. Once this amount is reduced to zero, aircraft can no longer be refilled from this refill station. Defaults to the amount of capacity

5.4.27.5 Using the module

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

Optionally, add a config zone and curtail airtank's behavior.

Optionally, add refill zones to the map.

5.4.28 Module Name

5.4.28.1 Description

5.4.28.2 Dependencies

5.4.28.3 Module Configuration

5.4.28.4 ME Attributes

5.4.28.5 API

5.4.28.6 Using the module

5.5 Server-based modules

Some modules in DML can interface with scripts that are installed on the server side of DCS, meaning that these modules usually only work when such a mission is run as multi-player.

For example, the module “ssbClient” interfaces with a popular server script (Ciribob’s “simple slot block”, “SSB”). If that SSB script is not installed on the hosting server, or the mission is run in single-player mode (i.e. it’s not run un a server), the mission module (ssbClient) will simply have no effect.

Other DML modules provide additional/strengthened stability to server-based missions, while other modules (e.g. “Tickle”) may even provide exclusive, server-only ability that aren’t available in single-player mode.

This section of the documentation describes the DML server modules: what they do, and how they communicates with a mission

5.5.1 smrGUI

5.5.1.1 Description

This tiny server modules allows a mission to re-start the mission simply by setting a flag to the value larger than 0

This is helpful for mission designers who incorporate the mission's version into the name (e.g. "my cool mission 1.2.7.miz") because a mission currently cannot easily restart itself. By installing the server module, a mission merely needs to change a flag, and the server module re-starts the mission.

This is also helpful during debugging sessions when you quickly want to re-start the mission without having to go through a mission editor cycle: Simply use the debugger (or some ME-built trigger) to trigger the restart.

5.5.1.2 Mission Interaction

smr watches the mission's flag space and reacts to changes in the following flags:

Flag	Description
simpleMissionRestart	When this flag has a value other than 0, smrGUI restarts the mission.

5.5.1.3 Using the module

The module must be installed in */Saved Games/DCS/Scripts/Hooks

5.5.2 stopGapGUI

5.5.2.1 Description

stopGapGUI provides some important functionality for the stopGap module on the server to work around a DCS synchronization bug. stopGapGUI has next to no performance impact and does not interfere with any other server module so you can simply install it on your server.

If your players complain that their aircraft are falling from the sky when they enter them, DCS is exhibiting the above-mentioned synchronization bug, and stopGapGUI is not active.

5.5.2.2 Mission Interaction

stopGapGUI provides required synchronization information for stopGap to prevent player aircraft from falling out of the sky when players enter the cockpit.

5.5.2.3 Using the module

The module must be installed in */Saved Games/DCS/Scripts/Hooks

5.5.3 noGapGUI

5.5.3.1 Description

Like stopGapGUI, noGapGUI provides some important functionality for the noGap module on the server to work around a DCS synchronization bug. noGapGUI has next to no performance impact and does not interfere with any other server module so you can simply install it on your server.

If your players complain that their aircraft are falling from the sky when they enter them, DCS is exhibiting the above-mentioned synchronization bug, and noGapGUI is not active.

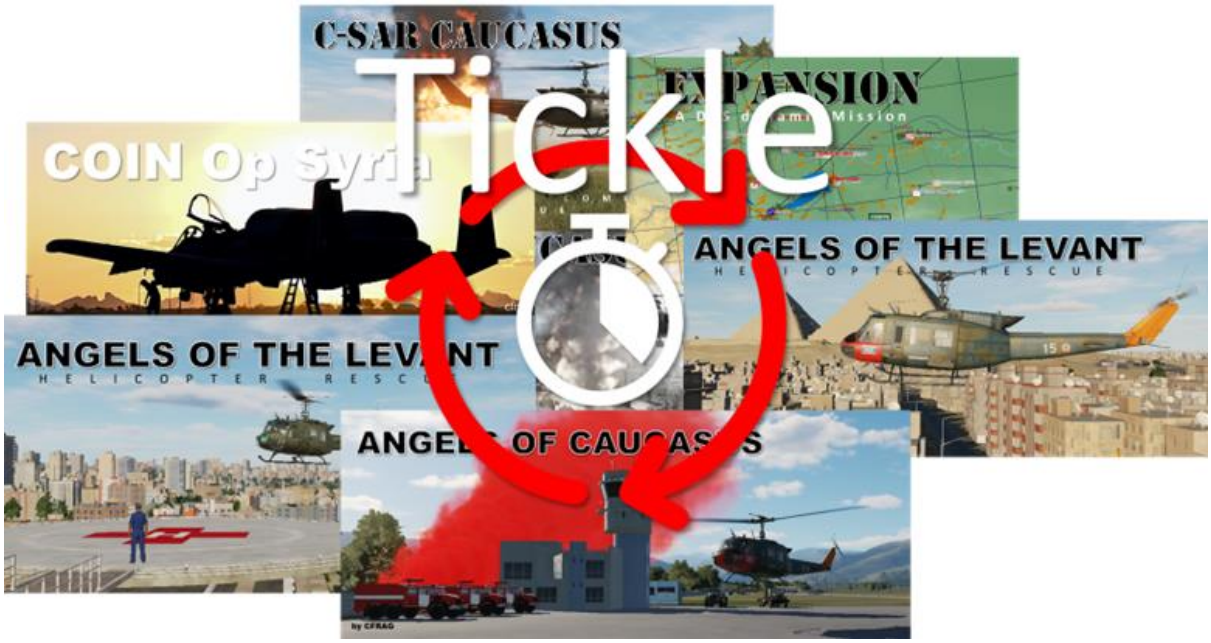
5.5.3.2 Mission Interaction

noGapGUI provides required synchronization information for noGap to prevent player aircraft from falling out of the sky when players enter the cockpit.

5.5.3.3 Using the module

The module must be installed in */Saved Games/DCS/Scripts/Hooks

5.5.4 Tickle (tcliGUI)



5.5.4.1 Description

"Tickle" is a tiny CLI ("Command-Line Interface" – the most primitive, yet also most efficient way to tell a computer what to do) that allows Eagle Dynamic's DCS servers to

- *automatically rotate* missions in your 'server playlist' on a time basis (e.g. after 4 hours) and
- allow *specially authorized* players to interactively (via mission chat) change the currently running mission

In other words, you can set up your server 'play list' with 5 missions, set the cycle time to 2 hours, and then the server changes the active mission every 2 hours to another mission from that play list. This can help you to keep your server 'fresh' for your players by serving up ever-changing missions automatically. For example, I set up a 'helicopter' themed server to cycle between six helicopter rescue sandboxes and change the active mission every 90 minutes.

Also, server owners (or players who are authorized) can change the currently playing mission simply by issuing a chat command (this requires the authorized person be on-line and inside the game).

Note:
Tickle is a server enhancement and only works in DCS multiplayer / dedicated server.

5.5.4.2 Mission Interaction

Tickle provides you (and all privileged users that you have added to the config file) with interactive chat commands that control the currently running mission.

Note: All Tickle commands start with a hyphen "-".

To issue a command in a mission, enter chat as authorized player, and type in a command as listed below:

- `-?`
  quick help and current automatic mission rotation status

- `-next`
  run the next mission in the server mission playlist. Loops. Ignores current shuffle setting

- `-prev` or `-previous`
  run previous mission from playlist. Loops. Ignores current shuffle setting

- `-restart`
  re-start this mission

- `-random`
  run a random mission from your server mission playlist

- `-pause`
  pauses the current mission. This stops mission time

- `-play`
  un-pauses the current mission. This continues mission time

- `-shuffle`
  Turns on mission shuffling, randomizing the next mission when the cycleTime timer runs down

- `-seq` or `-sequence`
  Turns off shuffling of missions when the cycleTime timer runs down.

- `-cycleTime`
  the big kahuna (Tickle's *raison d'être*): automatic mission rotation See "Automatic Mission Rotation", below

- `-cycle`
  Immediately cycle to the next mission (observes shuffle settings)

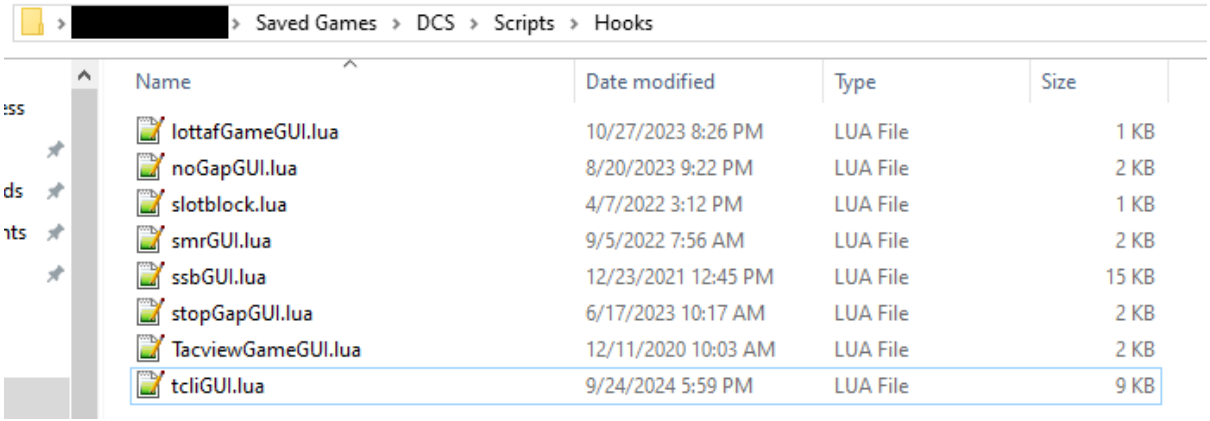
Whenever you (or any other authorized user) give Tickle a command in DCS chat, it is **not** broadcast, and only you receive any answers. This is intentional: for added security, nobody can see who gives what commands to Tickle.

5.5.4.3 Using the module

5.5.4.3.1 Installation

Installation is once-only, and drag-and-drop:

1. Put the file `tcliGUI.lua` into your DCS/Scripts/Hooks folder
2. Restart DCS

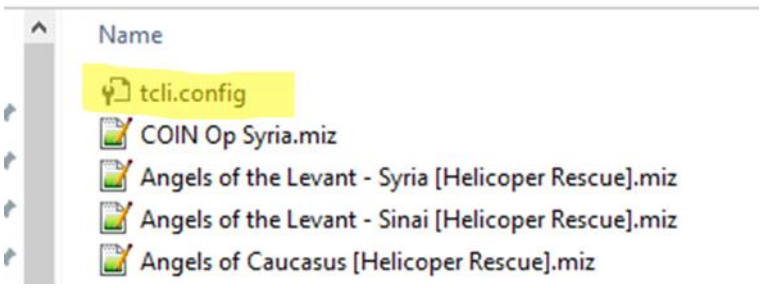


You now have installed Tickle on your computer. Its functions are only available in multi-player (local and dedicated server). If you start a multiplayer session now, you won't see any change because although Tickle is active, it does not yet know you.

If you have a hosting provider for your DCS servers (e.g. Fox3), you may need to have them put the `tclGUI.lua` file into the /Hooks folder for you.

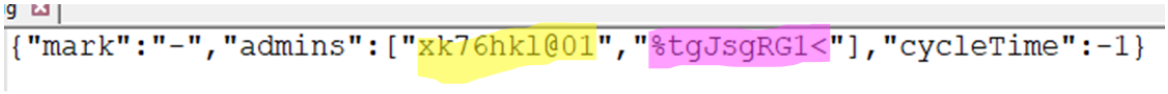
5.5.4.3.2 Authorizing players (interactive mode)

You need to tell Tickle who you are, and that it should accept your orders.



To do so, look into your **/Missions** directory and locate the file `tcli.config` (see the yellow markings in the image to the right). **It automatically appears in your /Missions as soon as DCS starts up after you install Tickle.** This config file contains the information that Tickle needs to recognize you (and other people whom you trust enough to control your server's mission sequence).

Use a text editing tool like "Notepad" or "Notepad++" to open *tcli.config*. There'll be some gibberish text that looks like this:



There are two silly privileged usernames pre-configured:

- xk76hkl@01 (marked yellow)
- %tgJsgRG1< (marked pink)

Replace one of them (or both) with **your DCS on-line user name**, and save the file. My DCS username is (obviously) CFRAG, and so, to authorize myself (and leave the second name untouched) on my DCS server install, my `tcli.config` file looks like this:

```
{"mark":"-","admins":["CFRAG","%tgJsgRG1<"],"cycleTime":-1}
```

i.e. I replaced xk76hkl@01 with CFRAG (you **must** leave the quotes (**"**) before and after the name). Save the config file and start a new server mission. Note that unlike creaky old DCS, Tickle notices, and loads your changes. It does **not** require you to restart DCS, all you need to do is start a new multiplayer mission. As soon as the server loads a new mission, Tickle reloads `tcli.config` from your /Missions folder. If you changed the config file correctly, Tickle now recognizes you as an authorized player.

Let's try this. Enter the mission, open a chat window and enter the command

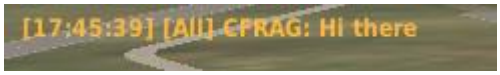
"`-?`" (without the quotes, return at the end)

If all went well, DCS responds (only to you) with a quick help screen that lists all commands, and tells you that currently, automatic mission changes are turned off.



If you don't see that response, you will have to re-edit the config file, save, and start a new multiplayer mission to reload `tcli.config`. Note that **usernames are case-sensitive** and therefore you must spell your DCS username exactly in the config file for Tickle to recognize you.

If you do not yet know your DCS username (it's the one you use to log into DCS), look at the chat window after you have typed a message. Your message always begins with your username.



Mine is CFRAG, all caps. Subtle, I know.

5.5.4.3.3 Automatic mission rotation

One of Tickle's main features is that it can automatically rotate missions on your server after some time (e.g. 2 hours) elapses. This feature is mainly for dedicated servers that run "endless" sandbox-type missions to automatically rotate though a roster of missions and provide greater variety to players. Tickle uses your server's mission roster and respects the "shuffle" flag (meaning: if shuffle is enabled in your server mission list, Tickle chooses missions in random order, else they are chosen in sequence).

To turn on automatic mission rotation, issue the command `-cycleTime` and supply the number of minutes each cycle lasts. For example:

`-cycleTime 240`

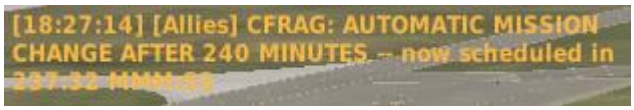


tells Tickle that it should change the currently running every 240 minutes (=4 hours).

Tickle automatically saves your cycleTime settings to storage and keeps them between DCS restarts. When you issue the `-cycleTime <amount>` command, that time interval is measured against the time when the mission started (i.e. it includes the time that already has passed in the current mission). So, if you are in hour 3 of your mission, and you give a cycle time of 120 (=2 hours), the mission cycles immediately.

To find out how much time is left in a cycle, issue the -cycleTime command without any parameters, e.g.

`-cycleTime`



If cycleTime is active and nearing the end, there is a warning broadcast 5 and 1 minutes before the mission changes.



To turn off mission rotation, provide a cycleTime of 0 (zero), e.g.

`-cycleTime 0`

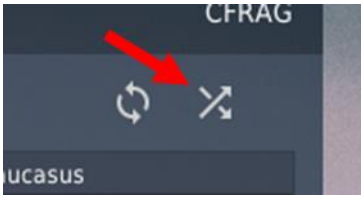
You can always check on the current cycle status by either giving the '-?' command or telling tcli '-cycleTime' (without a number).

Note:
Tickle automatically rotates to the next mission after sufficient **MISSION TIME** elapses. A *paused* mission freezes mission time, so when a server is set to automatically pause after the last client leaves, that server **will not rotate** missions unless clients return and the time finally runs down with players present. Set your server to "Resume on Load" to ensure that your server continues to run the mission after the last client has disconnected.

To rotate now, issue the `-cycle` chat command from within the mission.

5.5.4.3.4 Randomizing Missions

If you are local/ad-hoc-hosting your missions, you have the option to turn on ‘shuffling’ of missions on your playlist. If you



enable that option, Tickle automatically randomizes the next mission whenever cycleTime runs down.

WARNING:
Some of ED's GUI people appear to possess lamentably low UX skills. In the current ad-hoc server GUI metaphor, a *white* symbol signifies that the option is *on* (enabled), while a blue symbol means that it is off, disabled. It's just the way DCS rolls, I guess.

Some dedicated server web GUIs may not provide this option, making it inaccessible to dedicated servers.

Tickle to the rescue: you can interactively use the "`-shuffle`" and "`-sequence`" commands to turn on and disable shuffling, respectively. This is also saved to the server's configuration, so it will persist through server restarts, and of course also works for local/ad-hoc-hosted missions (here the GUI will reflect the current shuffle setting if you changed it interactively)

To turn off shuffling, issue the "`-seq`" or "`-sequence`" command interactively (or use the GUI for local/ad-hoc hosted missions)

5.5.4.3.5 Advanced Uses

Tickle supports multiple (unlimited even) privileged users. By default, Tickle provides you with two users in its `tcli.config` file:

["xk76hkl@01","%tgJsgRG1<"]

Inside those square brackets you can change the "names" inside the quotes. And you can add many more, simply by adding a comma "," and another name in quotes for each. For example, this is how the config looks if it was to support four users named "Iceman", "DingHe", "Suppa" and "Lil' me":

{"mark":"-","admins":["Iceman","DingHe","Suppa","Lil' me"],"cycleTime":-1}

PART IV:

PERSISTENCE

6 Persistence

This section deals with an optional DML feature called ‘persistence’. Standard DCS installations require that you actively disable security features to enable persistence. Many players do not feel comfortable with either the process of disabling these features, or the idea of playing with lessened security, or both. Therefore, I recommend that you think carefully before you add persistence to your mission. That being said, using persistence you can do incredibly interesting and fun things that are impossible to achieve otherwise.

6.1 What is persistence?

Simply put, persistence is the ability to read and/or write data to permanent store, so when a mission writes data and then ends, that data can be retrieved later and acted upon: the data persists past the end of the process. In other words, persistence is often used for:

- Saving and later continuing a mission (in DCS to a limited degree)
- Keeping records (scores, kills, departures, landings)
- Sharing data among different applications (or missions).
- Exporting data for other applications

This is an incredibly empowering ability for mission designers, as it theoretically allows a mission creator to write a string of missions that take into account the results or some main feature from the previous mission (say, units that were destroyed in the previous mission remain destroyed in the follow-up mission, making it more difficult or less difficult depending on the player’s past performance), and integrate it, creating the illusion of theater consistency.

So, why is persistency disabled by default and requires a non-trivial effort for the player to enable?

Security. Read on.

6.1.1 FOR YOUR OWN PROTECTION: READ THIS CAREFULLY

If a mission script has unfettered access to your storage, it can achieve a lot of great things. On the other hand, at the hands of a malicious mind, **it can also create a lot of damage**. Lua’s – and therefore DCS’s – only model of storage access is “all or nothing”. There are other, more granular, firewalled approaches that mitigate the risk of giving access to storage to applications (e.g. Apple’s Application Sandbox for iOS and macOS), but they are not available to mission designers in DCS. Since missions run as part of DCS, this means that you can either give a mission full access to your entire storage, or no access at all. So, if DCS gave full storage access to all Lua scripts, it will only take a few hours until the first maliciously crafted missions appear that exploit file access for nefarious purposes, with vandalism and blackmail being some of the *lesser* evils.

Therefore, by default, mission scripts can’t just willy-nilly write to your storage, you must explicitly instruct DCS to allow this. **That is good, as it protects casual players and mission designers who import a mission from the internet with malicious content from the damage that such a mission could cause.** This works well and makes DCS more secure.

The result, on the other hand, is that mission designers who want to write some data to file must actively disable these safety features before their mission scripts can write to storage. Again, this is a good thing, as it does not involve simply clicking a dialog (something a person may be socially engineered to do), but a more involved process that requires you to open and edit a file in DCS's main installation. In other words: If you allow DCS to write to your storage, that does not happen by accident.

More importantly, it also requires that anyone who runs (hosts) a mission that uses persistence must first disable these security features (this is only true for single-player missions that want to use persistence, and those people who self-host multiplayer missions. If you log on to a multiplayer game that uses persistence, your DCS does *not* have to persistence-enabled, all data is persisted on the server, never on your client).

Accordingly, if you author a mission that uses/requires persistence, be aware that this can significantly reduce the enthusiasm towards your mission from potential players. As a rule of thumb (and in DCS's current model of nothing/all access), persistency may be a good idea for long multiplayer missions and server missions that share player scores. It should be avoided for single-player missions, or be entirely optional.

6.1.2 De-Sanitizing LFS and IO

There are legitimate, even desirable reasons to allow a mission to be able to write to storage: to save and restore a mission, to manage high scores that persist through mission re-starts, to maintain a log of griefers, etc.

To use DML's persistence, **you (and anyone who hosts the mission that you are authoring) must** therefore perform the steps below to **"de-sanitize" two security features** that will remove restrictions place on Lua by DCS:

- Go to your main DCS installation (*not* the saved games folder, we aren't in Kansas any more)
- Open the folder named "Scripts"
- Look for the file "MissionScripting.lua"
- **Back up this file**
- Open the original
- Look for "`sanitizeModule('io')`"
- Place two dashes in front so it looks like this: "`-- sanitizeModule('io')`"
- Look for "`sanitizeModule('lfs')`"
- Place two dashes in front so it looks like this: "`-- sanitizeModule('lfs')`"
- Save and close the file, **restart DCS**

For reference, the block of code inside 'MissionScripting.lua' should look like this:

```
      sanitizeModule('os')
--    sanitizeModule('io')
--    sanitizeModule('lfs')
      _G['require'] = nil
      _G['loadlib'] = nil
      _G['package'] = nil
```

So, what have you done? By placing the two dashes in front of the '`sanitizeModule()`' commands you have commented them out, they will no longer be executed. The sanitize command disables libraries in DCS's scripting language "Lua". The two libraries that are disabled by 'sanitize', and that you are now allowing to run are:

- **io**
  This provides scripts with the ability to create, modify and delete files *anywhere* on your storage. An attacker might use this to read/modify/overwrite data if they know where to look (always assume that they do know this)

- **lfs**
  This provides scripts with the ability to create, modify and delete directories on your storage. An attacker might use this to create hidden, or hide existing folders, or change the structure of your storage and even delete whole directories.

Should anyone ever tell you to also de-sanitize 'os' be **very** suspicious. *That* library allows your scripts to spawn new processes, a capability that you should not easily give to other people nor scripts. That new process may be a virus, or key logger, or some other unsavory digital miscreant that you want to keep away from your computer.

Note:
Be advised that often, after a new DCS version comes out, you may need to repeat the de-sanitizing steps above, as the new version may overwrite MissionScripting.lua, putting the guards back in place.

6.1.3 Security Threat Assessment

Be aware that by de-sanitizing DCS, you are opening a window to allow an attacker onto your computer. A maliciously crafted mission script could then exploit access to lfs and io, however small that risk may be.

So how big is that risk? Risks are often measured in several categories: the *potential damage* that an event can cause, and the *likelihood* (probability) of that event happening.

So let's start with the bad one: Potential Damage is Catastrophic
If you de-sanitize lfs and io for DCS, and then run a mission that contains malicious Lua code, the potential result can be catastrophic. An attacker may destroy or modify any file on your storage. That's exactly why, by default, DCS disables lfs and io. So, let's be clear-eyed about that.

Now, for the better news: Likelihood is small
For the attack to succeed several conditions must be met:

- You must have lfs and io de-sanitized in your DCS installation
- A mission or mod with malicious code must be developed and posted online, with nobody detecting and warning about the attack code
- You must download that mission or mod that contains the malicious code and then run it in your unprotected DCS

The measures we talked about before make DCS an unattractive target for attackers: Since most DCS installations *have* sanitized lfs and io by default, the audience for such an attack is small. Although DCS is popular, it is not *that* popular (compared to, say “Minecraft”). Some 90% of all DCS installations are fully sanitized, most players do not require (nor ever notice the absence of) access to lfs and io. Hence, authoring such an attack vector promises a relatively small return on investment; there are bigger, easier targets. Just never assume that you are safe!

6.1.4 Mitigating Measures (if you are uncomfortable with persistence)

So, to mitigate the risk, you can try and follow best practices, some of which may be overkill (if, for example, you never run missions that you downloaded from unknown sources, then you can be more selective about your precautions):

- Be up to date on malware advisories, focusing on DCS.
- Obviously: if your computer/network is a high-value target, you should never de-sanitize lfs nor io. You should probably also not run games on it.
- Don’t run missions that you did not create yourself (often impossible)
- Only run third party missions or mods that you download from reputable sources that are peer-reviewed (e.g., ED, GitHub). Preferably, don’t be the first to run them.
- After downloading a mission, sanitize lfs and io before you run it for the first time, and keep an eye out on the internet for reports of DCS as an attack vector.
- Only de-sanitize lfs and io when you need to. There are third-party tools available that can automatically manage configuration files like MissionScripting.lua for you
- Re-sanitize when you are done. (Be advised: re-sanitizing cannot undo damage that may have happened while your DCS was open to attacks. That’s like closing the barn doors after the horse is gone))
- When you have a dedicated server, only de-sanitize the server.
- (If you are rich) use a dedicated mission development computer that is kept separate from the rest of your network.
- Make sure that your kids don’t subvert your efforts by de-sanitizing DCS to play that fabulous mission that they just found on the internet.

And don’t get paranoid. The threat is real, but it is entirely manageable.

6.2 Why persistence is currently difficult in DCS

Still with me and interested in persistence? Good. After the scary part above, let’s further curb your enthusiasm and look at what persistency can’t do due to the limits that DCS’s current game engine/scripting environment impose.

To be blunt: DCS is old. It’s based on the venerable ‘LOMAC – Lock On – Modern Air Combat’ originally published in 2003, which then evolved into DCS, originally released in 2008. So, by all accounts we are working with a game engine that is old enough to drink, even in the USA. At conception, the game’s mission engine wasn’t designed to host a theater containing hundreds of planes that are controlled by players in missions that span days of real-time. Accordingly, back then nobody thought it important that players be able to save mission progress to continue later.

Simply put: DCS wasn’t designed to support persistence. This currently (2024) has a number of consequences: You

- *Can't save time of day nor date*
  Technically, that's not entirely true. You *can* save time and date – you just can't set it in a mission when you load that information. This is one of the most significant impediments that can outright kill your mission idea. Imagine that you design a mission that starts at early morning 05:00 local time, and is intended to last for 16 hours, with the mission timed to take advantage of dusk falling in the last hour of the mission – an epic mission. If you play that mission and save it after 6 hours, and then later restore the mission from saved data, you may restore everything to where it was when you saved the mission – except that the time of day is back to 05:00 and that epic finale will not happen at dusk, but around mid-day. Worse, any flights that have their On-Target time at 1600 will still wait until it is 1600 game time, so they have de-synched from your mission and will be 6 hours late.

- *Can't restore weather*
  This is currently not much of an issue since dynamic weather isn't much of a factor in DCS right now. When you load a mission from file, the weather, like time of day reverts to the mission's starting conditions

- *Can't spawn AI units with less than full health, nor with depleted munitions*
  AI units currently can only spawn fully healthy, and fully equipped, with no mechanism to deplete them. This means that when you save a mission with units badly hurt and out of ammo, but not dead, then these units return to full health and fully equipped after the mission loads. So, beware of formerly depleted SAM sites when you load a mission – they are bound to be resupplied.

- *Can't spawn units halfway down their route*
  There currently exist no reliable method to spawn units with their task list restored from where they were when the mission was saved. This means that usually, after a mission restore, the units will attempt to return to their first waypoint, simply remain where they spawned after load, or some other unpredictable behavior (some of DML's modules can handle this gracefully, others not so much).
  The save behavior affects AI aircraft that spawn from the ground (take off): On load, they will always spawn on the ground, no matter how far they progressed at save point.
  So, design your savable missions with an eye on the consequences of adding complex movement paths.

- *Can't restore player's spawn and loadout*
  Similarly to AI units, player planes spawn at full health, with original mission payload and fuel, usually at their base. Some modules may support some clever alternatives, but with player planes there is the added difficulty that there is no guarantee that a player that was present when the mission was saved is also present in that same slot when the mission continues, so the simplest way to resolve this is to assume that that plane was lost, and a successor now starts at that slot's original point. From a mission designer's point, regard the mission's restore point simply as a player's log off /log on event.

And yet, even with all those restrictions, persistence still has a lot of interesting benefits, and here's how we can use persistence in DML.

6.3 Using Persistence in your Mission

To add persistence to your mission, simply add the 'persistence' module to your mission as the first module *after* dcsCommon and cfxZones.

IMPORTANT:
Add persistence immediately after you have added dcsCommon and cfxZones. **Persistence is only available to those modules that load after persistence** itself loads.

As soon as you add the persistence module to your mission, you have added the *ability* to load and save mission data. The persistence module itself can only save a pitiful amount of data by itself, all it knows how to do is the following:

- save the mission data that is handed to it and when it is told to save (including automatically every n minutes)
- load a mission upon start if a save file exists.
- skip loading mission data when told to, creating a fresh start.
- check mission save versions and ensure only valid data is loaded.
- save the flags that you tell it to

So, although persistence itself knows *how* to save data, it doesn't know *what* to save. That is provided by other modules. DML modules automatically converse with the persistence module, so you as a mission designer usually do not have to worry about that.

Not all modules support persistence. Some don't require persistence; others have not yet been upgraded to support it. Be sure to check a module's description if it supports persistence.

6.4 Dedicated persistence modules

DML provides a number of specialized persistence modules that can save the current status of mission 'items' that aren't under the control of other DML modules. Currently DML provides the following dedicated persistence modules:

- **unitPersistence**
  This module persists and loads all AI units that are placed with ME and are not under the control of DML modules. If such a unit is destroyed, unitPersistence makes sure that when loading a mission from storage, this module is removed.

- **WHpersistence**
  In late 2023, ED introduced a (quite shoddy, but still…) concept for mission-controlled warehouses. I'm still waiting for a warehouse concept that is worthy of that moniker (i.e. something that at least seems to be thought out rather than haphazardly slapped together by a sullen temp) so DML support for warehouses is currently severely limited. For the time being, DML does provide plug-in persistence support for warehouse inventories. So if your mission does manage warehouse inventories, simply dropping in the WHpersistence module will take care of saving and restoring them.

6.5 Using persistence with DML modules

In DML, modules either automatically support persistence, or they do not support it at all. If you want persistence, you add the persistence module. The rest happens automatically,

controlled via the persistence protocol that the persistence module implements and that other modules adhere to. Note that in some cases you may need to preserve a flag's status 'manually' by adding it via the *saveFlag* attribute to a trigger zones (see Persistence Module interface). This will cause the flag to be initialized to the saved value before the module starts up, making it the default value and prevent a trigger reaction when another module later changes it during its own load.

When a mission supports persistence, you can choose whether a mission should auto-save (in regular intervals), or only save on demand (via a flag change, of course). If a mission detects that it has saved mission progress, it will automatically start reloading from the saved data. This means that in order to start a mission from scratch, you must either tell persistence to ignore the save data (called a 'fresh start request) or delete the saved mission data from hand.

Persistence provides a mechanism for 'clean restart', and a ME-integrated 'update restart', which allows the mission to automatically restart from scratch if the save data is out of version with the current mission.

6.5.1 Restarting from scratch: 'Fresh Start'

To restart a mission from scratch, you can either delete the mission's save data (not very user friendly for dedicated servers) or issue a signal on 'cleanRestart?' for the persistence module as configured in the persistence modules config zone. Doing so writes a special value to the saved data that causes it to ignore any saved data on the next mission start. Usually, you would then follow this up with a mission restart.

6.5.2 Restart after Mission Update: versionID attribute

In persistence's config Zone (as set up in ME) you can provide a versionID attribute that is always saved with the mission data. When during mission start-up persistence detects that the current versionID does not match the one saved, it will start the mission from scratch. Use this if you release a new version of your mission that is incompatible with the data that a previous version has written, and thus must start from the beginning.

Note that if you are using versionID, persistence will write this data to a save file inside the save directory that you defined in the configuration zone.

6.6 Mission Loading: Sequence of Events

If you add persistence to your mission, most if it is automatic. For some modules it may be important to know in which order they are loaded so this is how persistence loads a mission

1) Persistence checks if it can access the mission save folder and if so, if there is any data. If either isn't true, persistence stops trying to load data. From this point on, the mission loads as if it was running for the first time: it starts 'fresh'
2) After reading the data, persistence checks to see if it finds the 'fresh restart' flag. If so, it stops loading data, and the mission begins fresh
3) Next, persistence checks if there is a version ID in either the current version (set via ME) or in the saved data and compares them. If they don't match (including matching an existing versus a non-existing ID), the mission begins fresh

4) From here on persistence knows that the data it is reading should be used to initialize all modules and signals (via flags and values) that data is available
5) It then proceeds to set all flags that were marked by 'saveFlags?' attributes
6) Persistence then relinquishes control to all other modules that load later. This means that no module that loads before "persistence.lua" can have any access to persistence, which is the reason why it should load as early as possible. **Note that the order in which these other modules load their data is indeterminate**, you must not assume or rely on the fact that one module loads data before another. **The only module guaranteed to load data first is persistence itself**.

Any module that is loading later proceeds as follows: first it

1) Checks to see if persistence is available. If so, it signs up for persistence save via a callback.
2) Then checks to see if persistence has data and requests the date by its module name
3) If it receives data, it proceeds to set its modules accordingly

6.7 Shared Data between Missions

Some DML module (e.g. Scribe) offer the ability to share data between missions. Usually, a mission writes its data into a self-contained file on your storage. When using data sharing, these modules automatically switch to shared data files that reside inside a "DML-Shared_Data" directory (usually also located in your main Missions directory). Using shared data is handles transparently by DML, and controlled with attributes, like any other ability that modules have.

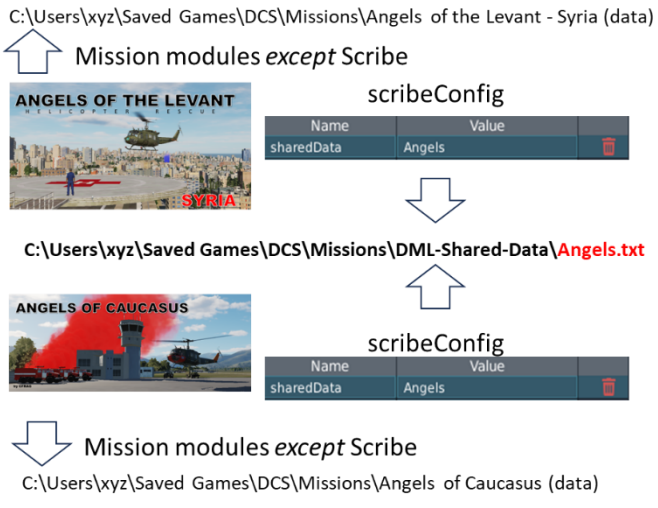
When you share data between missions, you provide a name for the shared data file, and the module retrieves its data from that shared file instead the mission's main data source.

For example, the missions "Angels of Caucasus" and "Angels of the Levant: Syria" both share the same data for their scribe module (and *only* their scribe module). Both missions have a 'sharedData" attribute in scribe's config zone with a value of "Angels". No other module except scribe uses shared data.

When either mission (*Caucasus* or *Levant: Syria*) loads, all modules load their data from their respective data files – except for the 'scribe' module, which accesses the shared data file "Angels.txt" in the shared directory "DML-SharedData".

More than two missions can share the same data, the same mission can use multiple different shared data files, and the same mission can have different modules write to the same shared file – persistence handles all this transparently and ensures that data is not inadvertently overwritten.

The image to the right illustrates the example: The mission *"Angels of the Levant – Syria"* uses its own (private) mission data file located at C:\Users\xyz\Saved Games\DCS\Missions\Angels of the Levant - Syria (data) \. This is where all modules read and write



their save data to and from. Likewise, the mission *"Angels of Caucasus"* has its own (private) mission data at C:\Users\xyz\Saved Games\DCS\Missions\Angels of Caucasus (data)\.

However, the **Scribe** module (and only the scribe module) uses its *sharedData* attribute to tell persistence that it wants to use the shared data file named *Angels* to read and write data from. This file is shared between all scribe modules that use the value *Angels* for their *sharedData* attribute, and is located at C:\Users\xyz\Saved Games\DCS\Missions\DML-Shared-Data\.

6.8 ME Integration

When you add persistence to a mission, you don't know if the computer that hosts the mission is de-sanitized (i.e., supports persistence) or not. From the mission's perspective, the computer always looks as if it *is* de-sanitized, the persistence module gracefully handles any exceptions internally. It therefore always looks to all modules as if they can save data successfully, even when they can't.

When it comes to loading data at a mission start, a secure DCS installation simply has no saved data, so the mission always looks to be a fresh start.

6.8.1 versionID

This attribute (which you can add to persistence config zone, see later) is used to automatically reset (discard) all saved data and start 'fresh' when the version found in the attribute does not match the value found in the saved data. You can use this feature to force a fresh start in a new version of your mission to avoid

6.8.2 Saving Flag Values

Persistence allows you to specify flags whose values should be saved and restored. To do so, simply add the "saveFlags" attribute to a zone. Use this to save flags that you have been using outside of DML.

Name	Description
saveFlags	A list of flag names to persist. Supports ranges for numbered flags. **MANDATORY**

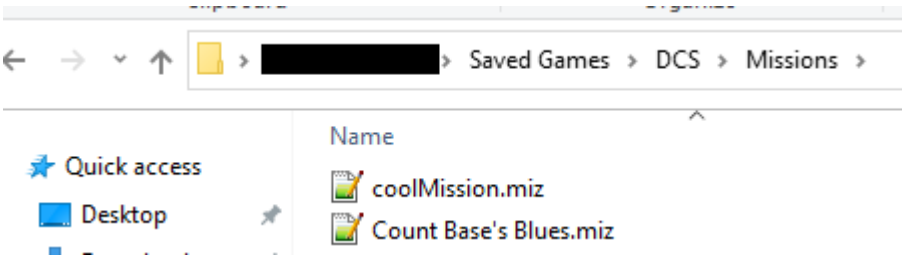
6.9 Configuration

6.9.1 Save Location

Persistence allows for easy integration with your mission, and defaults all optional parameters sensibly to ensure that – untouched- your mission data is saved and retrieved from its own location in your "Missions" directory. So, most mission designers do not have to worry about where persistence saves data – it simply works. The following is only required reading for people who change their default DCS installation – or those who simply like to know the additional details.

If you have customized your DCS installation to optimize storage, or configure a server in some non-standard way, rest assured that Persistence supports this as well.

So, here's how a fully defaulted mission named "**coolMission.miz**" would save in persistence, and how the various attributes work. First, we note that in a standard-configured DCS the mission is located in



C:\Users\xxx\Saved Games\DCS\Missions\

Since the mission is called "**coolMission**.miz", persistence would normally create the following structure to save all data:

C:\Users\xxx\Saved Games\DCS\Missions**coolMission** (Data)**coolMission** Data.txt

Meaning that it first creates a folder named "<missionName> (Data)" and then creates a text file called "<missionName Data.txt" inside that folder. This structure ensures that any later revisions to persistence can keep all mission data safely tucked away in it's on structure that should not conflict with any other mission data.

You can change all parts of the above by using persistence attributes "root", "serverDir", "saveDir" and "saveFileName" as follows:



In a fully defaulted configuration (i.e. out-of-the-box), persistence simply uses the following defaults:

attribute	**Default**
root	The directory that you configured DCS to be the 'home' directory. In a freshly DCS install that would usually be C:\Users\(user name)\Saved Games\DCS\ Defaults to what DCS tells persistence is the current home directory
serverDir	The directory name inside "root" that contains all missions. It's usually called "Missions\" and that is what persistence defaults to
saveDir	A folder (allocated if it doesn't exist) inside serverDir where persistence will save the mission data into a separate file. You can use this to pool multiple missions' data into the same folder. Defaults to "<mission name> (Data)" (see not below) **IMPORTANT NOTE** If you completely omit persistence's config zone, it reverts to simplified save mode, and defaults saveDir to "" (empty string), saving the mission's data file directly into the serverDir.
saveFileName	The name for the data (plain text in JSON format, can be edited with any text editor) file inside saveDir. Defaults to "<mission name> Data.txt"

Some modules (e.g. Scribe) support **sharing their data** with other missions. By default, these modules save their data inside the mission data file (see above). When enabling shared data, these modules write to a different file that is located at a different location on your disk, so that other missions can also access (read and write) that data.

Just like before, you can control where that data is written, and what name you give to that data file:



In the example above, the scribe module has a “sharedData” attribute that allows you to set the name of the data file that it looks for to load its data. All shared data files should reside in the same location (a recommendation only), and you can control where that file is going to be located.

attribute	**Default**
root	The directory that you configured DCS to be the ‘home’ directory. In a freshly DCS install that would usually be C:\Users\(user name)\Saved Games\DCS\ Defaults to what DCS tells persistence is the current home directory
serverDir	The directory name inside “root” that contains all missions. It’s usually called “Missions\” and that is what persistence defaults to
(immutable)	The name of the directory in which shared data is placed. This cannot be changed and is always “DML-Shared-Data”.
??? (module-specific data sharing attribute name)	The name for the data (a plain text-file containing data in JSON format, can be edited with any text editor) file inside “DML-Shared-Data”. To share data between missions, all that a module must do is use the same value for the sharing attribute in different missions. Note that the name for the data sharing attribute is not controlled by the persistence module but the module that owns the data (e.g. *Scribe*) Multiple modules can use the same data file inside DML-Shared (e.g. the modules *Scribe* and *xyzUnknown*), so you do not have to create separate data files for each module inside shared data. If a module uses the same file name in shared data in multiple missions, these missions all access the same date file across multiple missions. *Example:* The *Scribe* module supports a “sharedData” attribute to set the name of the file used in the shared data folder. If you use the value “Angels” for this attribute *in multiple missions* that use scribe, these *scribe* modules incarnations (one per mission) share their data between all missions that use “Angels” as value for this attribute, and the file “Angels.txt” inside “DML-Shared-Data” holds this shared data.

6.9.2 'Unconfigured' persistence

Persistence can be configured with a config zone. If you do not provide such a config zone in your mission, persistence will default to simplified behavior

- It saves all data to the "Missions/" directory (i.e., **not** "<mission name> (data)/")
- No version handling (obviously, as this is set in a config zone)
- Since no cleanRestart? Input is defined, you must either invoke the method via API, or manually delete the data file to force a fresh mission start
- saveInterval is set to -1, so persistence only saves data on manual invocation via API (since saveMission? isn't set)

6.9.3 Persistence config zone

You can configure the persistence module using a trigger zone in ME named "persistenceConfig" and add the following attributes

Name	Description
verbose	A value of "true" turns on debugging messages. Default is "false"
versionID	If present, this turns on version matching. When a mission starts up, persistence checks the value provided via the Zone with the one saved. If they do not match, the entire save data is discarded, and the mission starts fresh Defaults to <none>
root	Path to the DCS standard directory (usually "C:\userName\saved games\DCS.openbeta\" or "C:\userName\saved games\DCS\"). This value is passed from DCS to persistence for default. You can change this to adapt your missions to conform with more elaborate server setups. If you change this. Be sure that you know what you are doing, and initially have verbosity set to true, so you can see which directory your mission will save to. Defaults to your currently running DCS instance's write dir.
serverDir	Path from the root directory (see above) to the Missions directory. Use this if you set up your DCS different (usually important for dedicated servers). Defaults to "Missions\"
saveDir	Name for the mission's data directory. Defaults to "<mission name> (data)". This directory is created in the serverDir automatically if it does not exist If you set saveDir to "", the mission saves its data directly into serverDir Defaults to "<mission name> (data)" if a configuration zone is present, none without configuration zone (i.e. the data is written into serverDir)
saveFileName	Name for the file that persistence uses to write mission data. Defaults to "<mission name> Data.txt"
saveInterval	Controls auto-save. Any value larger than zero will turn on auto save. The value you give here is the number of minutes between auto saves. Auto-saves co-operate with manual saves, so you can use both methods in your mission Defaults to -1 (auto-save off)
saveNotification	When set to true, each time that the mission is persisted, a text

Name	Description
	notification is sent to all players. Default is true (notify players when saving)
cleanRestart?	DML Watchflag. A change signal on this input triggers a “fresh start” request: next time the mission starts up, it won’t load mission data. Defaults to <none>
saveMission?	DML Watchflag. A change signal on this input triggers a ‘manual’ save. Defaults to <none>

PART V:

THE DEBUGGER

7 The Debugger

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



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

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

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

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

Plain-vanilla missions use *flags* as their method to control mission flow. In non-DML missions, designers place *trigger zones*, and then create *trigger rules* to perform *actions* when (and only when) certain *conditions* are met. Once conditions are met, a common result is to change a flag to remember that a condition was met.

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

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

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

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

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

- **Set-up / Design Mission** – this is while you create your mission in Mission Editor. Whenever you feel like it, you can add attributes to trigger zones that tell the debugger about things to watch out for. You can set up your debugging session in your own time in Mission Editor, and have all information ready at your finger tips when you start the mission

- **Execution / Run Mission**– this is *while your mission is running*. Here The Debugger comes into its own, and hands you an array of powerful tools to hunt down even the most obscure of bugs. Ordinarily, you would only be able to observe. With The Debugger you can interactively intercede.

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

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

DML-Version Only

- Cross-reference flags:
    - used as inputs? and outputs! for which zones
- Cross reference DML modules (zones)
    - connect to which flags (inputs and outputs))

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

7.1 Using The Debugger (Summary)

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

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

7.1.1 Adding The Debugger to your mission (DML)

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

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

7.1.2 Adding The Debugger STANDALONE to your mission

Add “Mission Debugger.lua” as a DOSCRIPT or DOSCRIPTFILE to your mission AT START.

7.1.3 During design (Mission Editor)

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



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

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

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

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

7.1.4 In-Mission

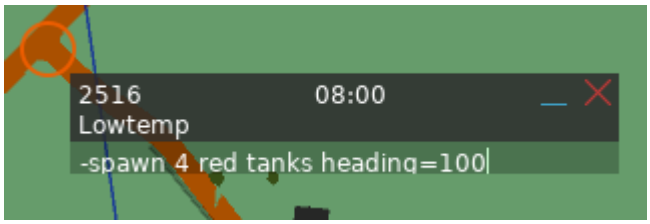
The Debugger is **interactive**, meaning that at any time while the mission is running, you can interact with it: tell it to track flags, inspect tables, spawn or remove units etc. DCS was not designed for that kind of interactivity (it lacks what people call a 'console' – an interface to enter commands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

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

- The mission starts up, The Debugger loads, and sets up accord to any attributes that you may have added to trigger zone with Mission Editor.
- It watches events and flags as instructed and tells you when something happens that you have told it to watch out for



- The Debugger also watches whenever a player enters something into the "Mark Label" game function, and if it matches one of its command keywords, it springs into action, executing the command that the player has invoked. These commands can range from adding or removing flags at and from watch lists, so spawning units, or even changing "tables" in the mission scripting environment – ad hoc changes to the mission



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

You would normally use The Debugger to

- Verify that things happen in the order that you have envisioned while you designed the mission. For this you monitor flags and tables to ensure that they change when, and in the way, that you have anticipated.
- Artificially force situations that you may have not foreseen. For this you change flags, spawn units or remove units.

- Check what happens (the sequence of things that happen) when a certain situation arises – again by monitoring flags and tables.
- Make situations arise (very useful to test otherwise difficult to test for edge cases). For this you change flag values, set table values, spawn units or remove units at will.
- Investigate and document how DCS handles certain situations to make sure that your assumptions are valid, and that your mission handles this correctly.
- If you have enabled persistence, you can also export the debugging log to a text file for better analysis later.

7.1.5 Debugger commands (overview)

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

7.1.5.1 Changing and Observing Flags

Command	Description

show	Syntax
	
	`show <flagname/observername>`
	
	Display the current value of all flags that are observed by <observername>. If there is no observer <observername>, the value of the flag named <flagname> is displayed.
	
	Examples:
	
	`show samStatus`
	
	Displays the current values of all flags that are observed by the observer “samStatus”. If there is no such observer currently defined, the value of the flag named “samStatus” is displayed.

set	Syntax
	
	`set <flagname> <number>`
	
	Set the flag named <flagname> to the numeric value of <number>.
	
	Examples
	
	`set samsAlive 1`
	
	This sets the value of the flag named “samsAlive” to 1 (the number one)

inc	Syntax `inc <flagname>` Add 1 (one) to the flag named <flagname>’s current value.

Command	Description
	Examples `inc samsAlive` This adds 1 to the current value of the flag named samsAlive. If the current value is 5, then the new value will be 6

flip	Syntax
	
	`flip <flagname>`
	
	Set the value of the flag named <flagname> to either 0 or 1. If the current value is 0, then the new value is 1. If the current value is anything but 0, the new value is 0.
	
	Examples
	`flip samsAlive`

observe
o
	Syntax
	
	`observe <flagname> [with <observerName>]`
	
	Add the flag named <flagname> to the observer <observername>. If you omit <observername>, the flag is added to The Debugger's default internal observer.
	Flags that are under observation are checked periodically for a change in value. Should the value change, The Debugger reports the change.
	
	Examples
	`observe samsAlive`
	`o samsAlive`
	
	Adds the flag samsAlive to the internal observation list

forget	Syntax
	
	`forget <flagname> [with <observerName>]`
	
	Remove the flag named <flagname> from the observer <observername>. If you omit <observername>, the flag is removed from The Debugger's default internal observer.
	
	Examples
	`forget samsAlive`
	
	Removes the flag samsAlive from the internal observation list

new	Syntax `new <observername> [[for] <condition>]` Create a new observer named <observername> that you can add flags to observe to. If you omit the "with <condition>" part, the

Command	Description

	observer is created to watch for “change” in the flag’s value. The Debugger understands all current DML Watchflag conditions
	
	Examples
	`new watchingU with >3`
	`new watchingU >3`
	
	Creates a new observer named “watchingU" that is set to trigger if a flag’s value changes, and the new value is greater than 3 (three)

update	Syntax
	
	`update <observername> [to] <condition>`
	
	Change the condition that trigger flags watched by the observer named <observername> to <condition>.
	
	Examples
	`update watchingU to change`
	`update watchingU change`
	
	Changes the trigger condition for the observer named “watchingU" to “change”, i.e. any change in value

drop	Syntax
	
	`drop <observername>`
	
	Remove the observer named <observername>
	
	Examples
	`remove watchingU`

list	Syntax
	
	`list <match>`
	
	List all observers whose name contains <match>
	
	Examples
	`list ching`
	
	Lists all observers that have “ching” in their name, e.g. “watchingU”

who	Syntax `who <flagname>` List all observers are tracking (observing) the flag named <flagname> Examples

Command	Description
	`who samsAlive` Lists all observers that (perhaps among other flags) observe the flag named “samsAlive”

reset	Syntax
	
	`reset [<observername>]`
	
	Re-load all values of the flags that are currently observed by observer <observername>. If you omit <observername>, all observers are reset. This command is useful prior to re-starting The Debugger when you have suspended it during testing, as it reloads all flags at their current values, avoiding false positives when The Debugger resumes.
	
	Examples
	`reset watchingU`
	
	Lists all observers that have “ching” in their name, e.g. “watchingU”

7.1.5.2 Analysis

Command	Description

snap	Syntax
	
	`snap <observername>`
	
	Create a “snapshot” of all flag values that are observed by observer <observername>. This snapshot can later be used to compare to current values to see which flags have changed.
	If you omit <observername>, a snapshot of all currently observed flags is created.
	
	Examples:
	
	`snap watchingU`
	
	Creates a snapshot of all flags that the observer “watchingU” is observing.

compare	Syntax `compare` Compare all flags from the last snapshot with their current values and provide a table with past and current values; flags that have changed in value are marked Examples: `compare`

Command	Description

note	Syntax
	
	`Note <any remark>`
	
	Send <any remark> to The Debugger's output, allowing you to annotate it (useful only if you plan to export the output for later analysis)
	
	Examples:
	
	`note <-- before landing of A-10`
	
	Inserts "<-- before landing of A-10" into The Debugger's log

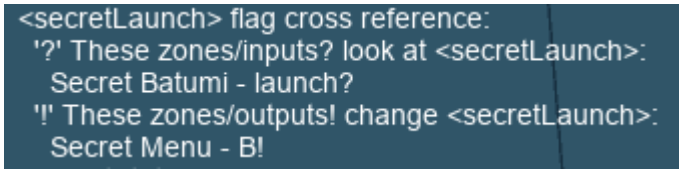
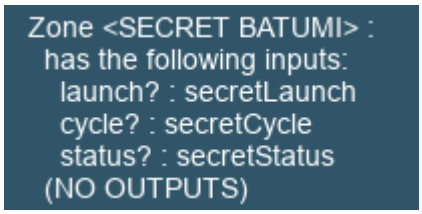
save	Syntax
	
	`save <file name>`
	
	Save the contents of The Debugger's log to a plain text file with the name "<filename>.txt"
	
	Examples:
	
	`save Sinai Tourist Test`
	
	Requires that
	• The **persistence** module is loaded and
	• Your **DCS** installation is **de-sanitized**
	If writing fails or the persistence module is missing, The Debugger will complain, and then continue working as if nothing happened.

7.1.5.3 DML integrated Analysis (Zones, Inputs, Outputs)

Command	Description

x	Syntax
	
	`x <flagname/zonename> | ? | * | *f | *z`
	
	Gives you a comprehensive **cross reference** about which zones use which flags as inputs, outputs, or direct flags, or which flag connects to which zones.
	
	Examples:
	
	`x ?`
	
	Displays a short summary on how to use the cross reference function.
	
	`x *`
	Provides a comprehensive cross reference of all flags and zones

Command	Description

	that are used in this mission in a DML context as inputs or outputs.
	
	Note:
	This full cross reference is often large – too large to fit on-screen. Use DCS's "Messages History" function or the debugger's save as text ability to browse the output at your leisre.
	
	`x *f`
	Provides a comprehensive cross reference of all flags that are used as in- or outputs for DML modules. It lists all flag names, and differentiates which zones use the flag as input (?), and which use it as output (!, #) along with their attribute names
	
	`x *z`
	Provides a comprehensive cross reference of all DML zones and the flags they use as inputs (?) and outputs (!, #). It lists all zones and provides separate information in inputs and outputs. For each input/output the attribute name is shown, and the flag names that connect to it
	
	`x <flagname/zonename>`
	When cross reference is invoked with a **flag name**, this provides a cross reference of which DML zone uses this flag, and for each zone that uses this flag, it also provides a list of all inputs ('look at') and outputs ('change') that use this flag, along with their attribute names:
	 
	
	When cross reference is invoked with a zone name, this provides list of all inputs (?) and outputs (!, #) of that zone, along with the flags that are used
	
	 
	
	If <flagname> isn't used as an input or output for a zone, and <zonename> isn't the name of a zone, an error message "NOT USED WITH DML" is displayed

7.1.5.4 Spawning and Removing Units, Objects or Effects

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

Command	Description

spawn	Syntax
	
	`Spawn [<number>] [<coalition>] <type> [heading=<number>]`
	
	Spawn <number> instances of <type> belonging to <coalition> into the game. The spawned units have a heading of number. If you omit
	• <number>: 1 (one) unit is spawned
	• <coalition>: the units belong to NEUTRAL
	• heading=<number>: the units head 000 (North)
	
	To get a list of all allowed types and their current settings, use the special command
	
	`spawn ?`
	
	Examples:
	
	`Spawn 4 red tank`
	
	Creates 4 tanks belonging to RED at the position of the Mark Label, heading North (000). The units that are created for the “tank” type are defined by the tape name that you can pass to The Debugger with a trigger zone; by default, it is a “T-90”. To see which type is currently set to what, use the `‘spawn ?’` command

remove	Syntax
	
	`remove <name>`
	
	Remove the group/unit/object named <name> from the game. If you don’t know the name of an object, unit or group, you can use the F-10 map and click on it.
	
	Examples:
	
	`remove Lander-1-1`

smoke	Syntax
	
	`smoke <color>`
	
	Add smoke with the color <color> at the location of the Mark Label. The colored smoke times out after 5 minutes
	
	Examples:
	
	`smoke red`

boom	Syntax `boom <number>`

Command	Description
	creates an explosion of strength <number> on ground level at the point of the Mark Label. If you omit <number>, an explosion of strength 1. Note that a strength of 3000 is usually sufficient to level a block of buildings in a town. Examples: `boom 20`

7.1.5.5 Events

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

Note that events are an advanced mission design topic.

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

7.1.5.6 Inspecting and Changing Tables

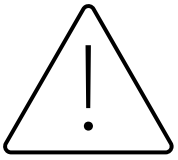
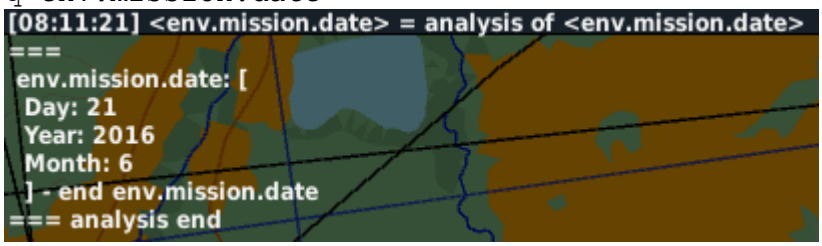
While flags and events make out some 99% of most mission designer's mission debugging work, some designers also need to get into the nitty gritty of design, and require access to the deeper abyss of the mission scripting environment – they want to inspect (and possibly change) tables.

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

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

Command	Description

q	Syntax
	
	`q <fully qualified table name>`
	
	Return the value of a Mission Scripting Environment table.
	
	Examples:
	`q env.mission.theatre`
	
	Returns the value of the env.mission.theatre table. With standard missions, this returns the name of the map (a string) that the mission is run on, e.g. “Caucasus”.
	If the type of the queried table is a number or string, the value is returned immediately, otherwise the type is returned. You can then use the ‘a’ (analyze) command to get a detailed description of the table’s content.

a
 	
 
	Syntax
	
	`a <fully qualified table name>`
	
	Analyze a Mission Scripting Environment table. Recursively analyse the entire table and visualize the structure as text. Some tables are really big, and The Debugger is not easily frightened, so beware telling it to analyze, for example, “env”. Because it will.
	
	Examples:
	`q env.mission.date`
	 
	
	Analyses the structure and lists all node values of the env.mission.date table.

w
 	
 
	TREAD CAREFULLY – THIS IS NOT A DRILL
	
	Syntax
	
	`w <fully qualified table name> [=] <any Lua expression>`
	
	Assigns (and executes if required) <any Lua expression> to <fully qualified table name>. <any Lua expression> can be anything legal in Lua, including full programs.
	
	Examples:
	`w dp = {x=1, y=2}`
	
	Creates a table with fields x = 1 and Y = 2 and assigns it to the new

Command	Description
	global table dp.

7.1.5.7 Miscellaneous

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

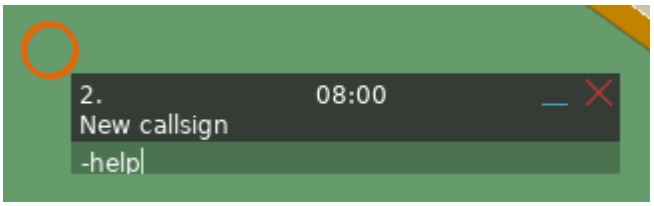
7.2 Interactive Use during Missions

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

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



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



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

The debugger instantly responds with a text message:

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

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

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

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



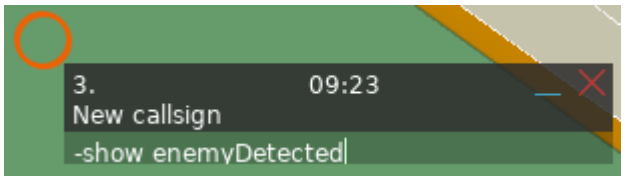
We'll go through all the commands below, so there's no need to study above; just nod in appreciation.

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

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

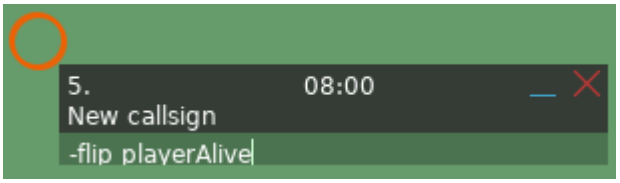
The Debugger has three easily understood commands for that:

- **-show <flagname>**
  Accesses the flag <flagname> (e.g., 'enemyDetected') and displays its current value as a text message. Note that all debugger messages are time-stamped so you can (using Message History) reconstruct when (in mission time) the flag had that value.:




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

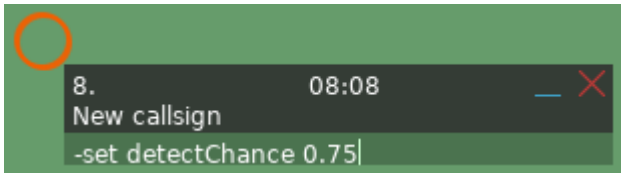
- **-inc <flagname> and -flip <flagname>**
  Most of DML's module inputs react to a *change* in a flag's value, meaning that mostly, they aren't looking for a particular value, merely a *change* in value. To conveniently support this, the debugger sports two quality of life features: commands to increment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




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

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

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



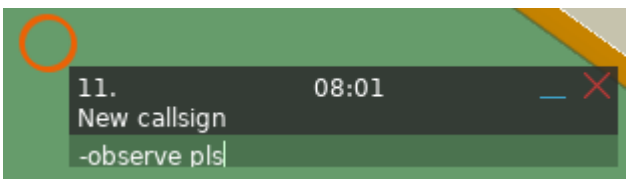

```
*** [08:10:07] debug: set flag <detectChance> to <0.75>
```

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

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

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

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



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

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

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

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

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

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

The debugger tells you:

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

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

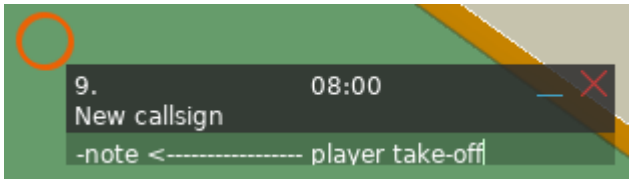
7.2.4 Losing interest: -forget



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

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

7.2.5 Annotating Message History: -note



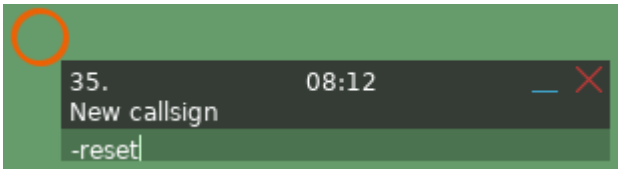
A lot of debugging has to do with looking at the message log (Message History) and finding the relevant entries. Thus, it may be helpful to find points of interest that you

marked: a flag value that you found strange, or an event that you want to mark in the log and later find. For this, the debugger has a '-note' command. Anything you type after the command will appear as a text message in the log, making it easy to find later:



7.2.6 Clearing the Slate: -reset

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



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

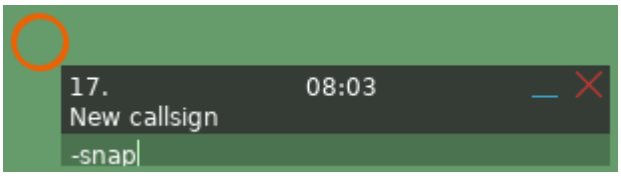


7.2.7 Getting the BIG picture: -snap and -compare

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

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

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

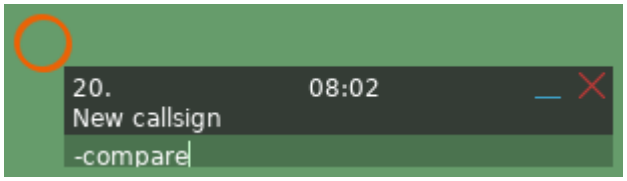


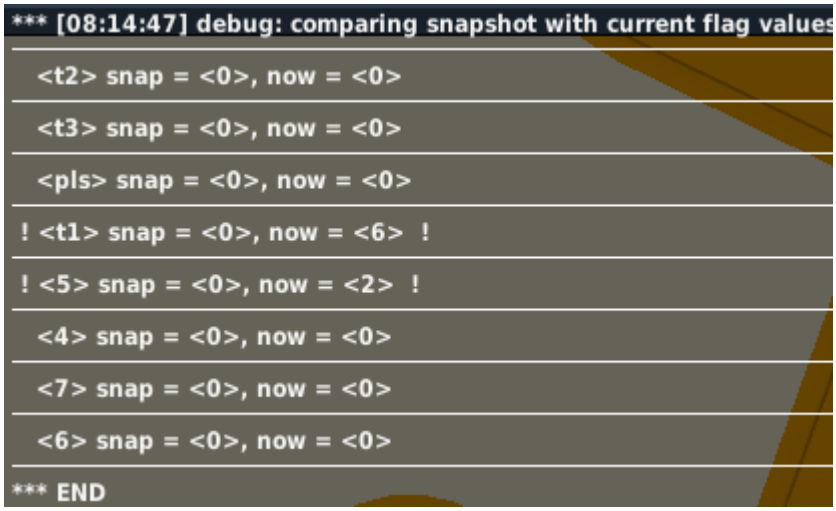


Note that the debugger's response tells you how many flags it is tracking in the snapshot. Make sure that you check this against your expectation, as there are few more frustrating situations when you find out that that one flag you needed to compare wasn't part of the snapshot.

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

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





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

7.2.8 Getting cross with you: details on zones and flags: -x

DML modules provide mission designers with a plethora of cool functionality that they can connect with flags to control mission flow, and that allow them to create complex missions with a few simple clicks: place some trigger zones and add some attributes. Unfortunately, with growing abilities usually comer greater ambitions, and soon missions can become so complex that some smaller detail can get lost or become forgotten: which flag caused all cloners to spawn, and did I perhaps forget to connect one of them?

The Debugger comes with a powerful, dedicated tool to help you manage DML-specific topics: **Cross Reference**. It allows you to instantly get information on which flag connects to which DML zone, and it gathers for you which zone uses which flag – and how.

7.2.8.1 Where and how is a flag used with DML?

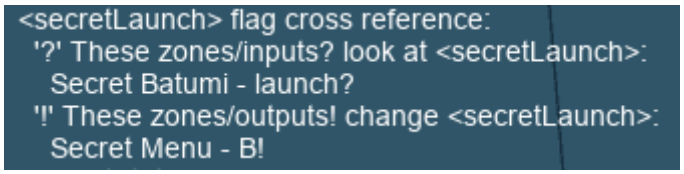
Say that you have a flag named 'secretLaunch' that you intend to use in your mission to secretly launch something nasty at your players. Now, during playtesting, something doesn't

seem right, and you wonder if either some other zone changes the flag, or if perhaps you did not connect this flag correctly to the launcher.

```
-x secretLaunch
```

quickly gives you the answer:

It first lists all zones that are looking at this flag with an input? attribute, and lists the zones and their attributes. In our case, the DML Zone named ‘Secret Batumi’ has an input? named ‘launch?’ that is looking at this flag. In other words, the flag secretLaunch connects to the ‘launch?’ attribute of the DML Zone ‘Secret Batum’.



Next, the cross reference lists all DML zones and attributes that can change the flag named ‘secretLaunch’, that have an output! connected to this flag. We see that the DML Zone ‘Secret Menu’ has an attribute named ‘B!’ that connects to ‘secretLaunch’ and can therefore change the value of ‘secretLaunch’

Looking at this differently we can say that Secret Batumi’s “launch?” input is controlled with Secret Menu’s “B!” output.

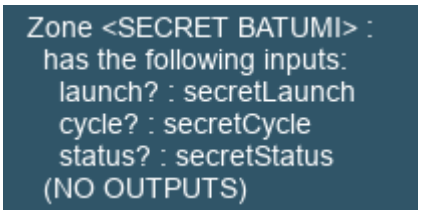
7.2.8.2 Who (which flag) connects how to my DML zone?

Let’s turn that around and find out what flags connect to my trigger zone “Secret Batumi”.

```
-x secretBatumi
```

Now you’ll get a list of all the flags that connect to DML zone’s inputs and outputs:

In our case we see that there are three input? attributes for “Secret Batumi”, and each of them connects to a different flag: ‘launch?’ looks at (connects) to the flag named ‘secretLaunch’, ‘cycle?’ looks at the flag named ‘secretCycle’, and the input ‘status?’ looks at the flag named ‘secretStatus’.



Furthermore, there are no outputs active for this zone (“NO OUTPUTS”) – if there were, we’d again see a list of their attribute names and the flag names that they connect to

So what happens if you have a flag that has the same name as zone? Don’t try to find out. Make sure that zones and flag names spaces are always distinct and have no overlap.

But if you positively, 100% must know: you’ll only get the information about the zone.

7.2.9 Finding bad ‘unconnected’ flag names / commands: -x !

One of DML’s most powerful abilities is that modules talk to each other. They do this by using flags – or, when you are using the “Yeller Model” – commands. There’s a small but nasty catch, though: when you use a flag/command to connect an input to an output, it is incredibly important that you always spell the command/flag exactly for both output and input attributes. Just a single tiny difference, a missing letter or different capitalization prevents

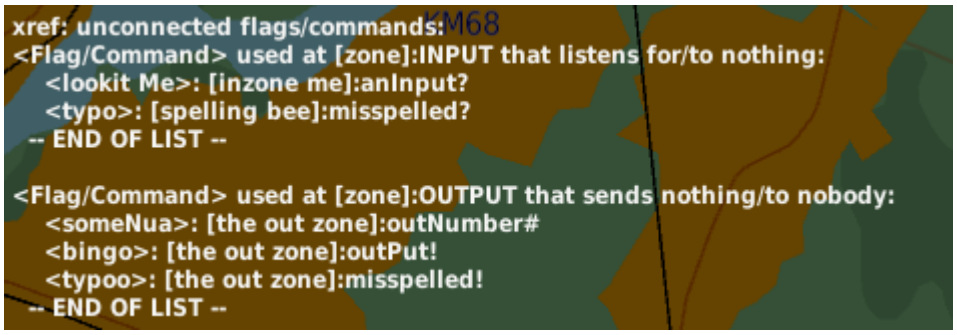
input and output from finding each other. A single, small typo can thus have big consequences.

The Debugger has its own dedicated function to track down exactly this issue. It can do that by looking at all inputs and outputs and noting down the commands/flag names. And then it uses a simple statistical hint: if a flag or command is only used on one side (input or output), there is a good likelihood that it was misspelled. This usually means that misspelled commands happen in pairs: one for each side. In DML we speak of “mismatched pairs” – the input doesn’t pair with the output as intended. The Debugger finds these pairs, and shows them with the zones that they are used in.

The debugger command

```
-x !
```

calls up this function, and always displays all mismatched pairs for you to easily track down these typos.



You should make it a habit to always check your mission for mismatched pairs when you start your debugging, there’s a good likelihood that you find the culprit to your issues quickly with this cross reference ability.

7.2.10 Bringing in the (big) Guns: -spawn

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

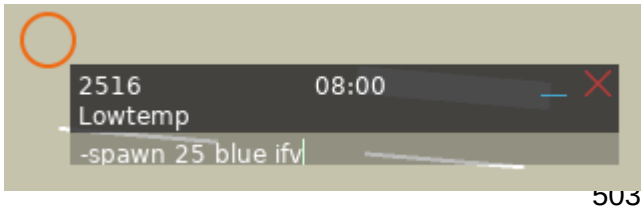
7.2.10.1 The spawn command

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

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

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

- *how many [default = 1]*
  A number that tells The Debugger how many units of <type> it should spawn. If you spawn ground or naval types, there is no



strict upper limit, and theDebugger will happily spawn 1000 units for you. If DCS can then handle the strain is something else entirely.



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

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

- *faction* [default: NEUTRAL]
  The faction (red/blue) to whom the spawned units belong.

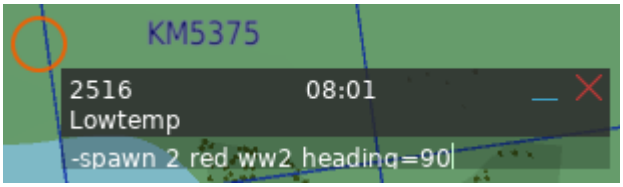
- *heading=0 [default:0]*
  No spaces, the word 'heading' and the equal sign are mandatory. The heading is heading in degrees, with 0 being North.
  If present, all units will head in that direction. If omitted, all units will head North. If the spawned units are aircraft, they receive also receive a waypoint 100km in that direction.

- *type*
  This is spawn's greatest ability. There are literally hundreds of different units in DCS, and it would be silly to assume that mission designers know their correct (arcane) type designation by heart. So The Debugger uses a system of **Generic Stand-ins**: you say 'tank', and The Debugger says 'T-90' (with "T-90" being DCS's correct internal unit typeString for a T-90 main battle tank). So, all you need to remember is the Generic name (see 'Spawned Generics, below), and The Debugger looks up the correct DCS unit typeName for you. Even better, you can tell The Debugger which in-game types to use for which generic name.

7.2.10.2 Spawned Generics

The Debugger knows the following generic names:

- *tank – default T-90*
- *ifv – default BTR-80*
- *inf* [infantry] – *default Soldier M4*
- *sam – default Roland ADS*
- *aaa – default Zoo Shilka*
- *arty* [artillery] – *default M-109*
- *manpad – default Stinger Manpad*
- *truck – default KAMAZ Truck*
- *jet – default Mig-21*
- *ww2 – default Spitfire*

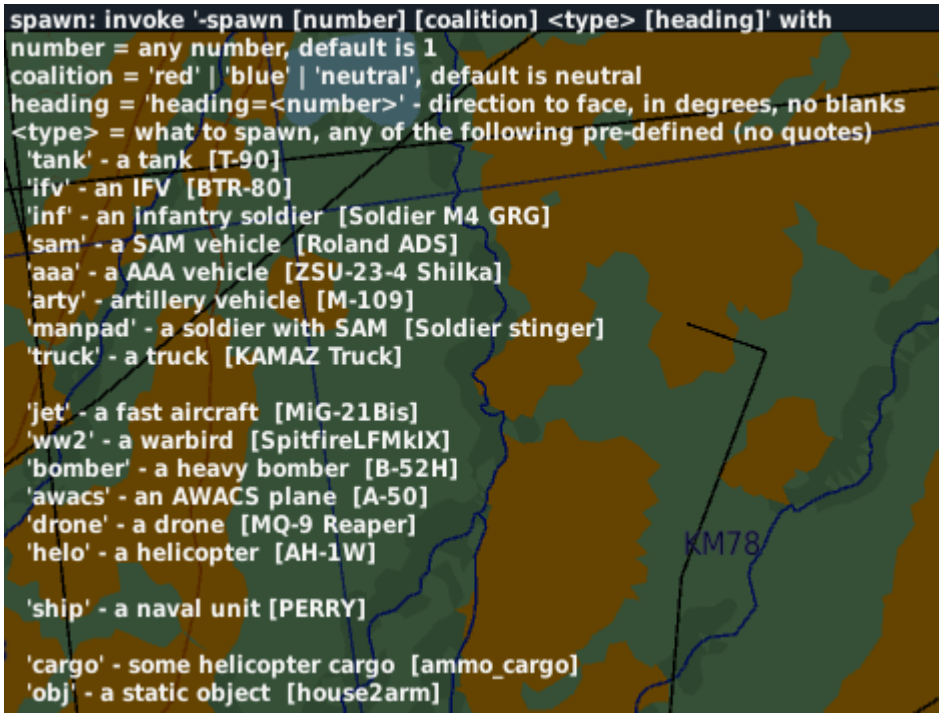




- *bomber – default B-52*
- *awacs – default A-50*
- *drone – default Reaper*
- *helo – default AH-1*
- *ship – default Perry*
- *cargo* [for helicopters] – *default ammo*
- *obj* [static object] – *default Armed Watchtower*

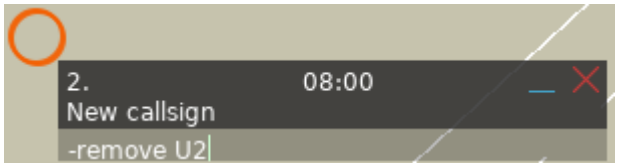
And if you can't remember which generics are available (and which in-game they are currently configured to spawn), you can always issue the *spawn help* command:

```
-spawn ?
```

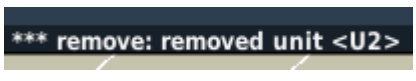


7.2.11 Getting rid of 'em: -remove

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



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



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

7.2.12 Smoke On! -smoke

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

7.2.13 Boom Baby! - boom

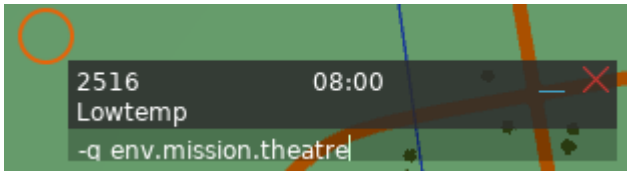
Similarly, you can place explosions with the -boom command. Except a smoke color, add an explosion strength (a number), and there's a big boom at the location of the Mark Label.

7.3 Killer Features

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

7.3.1 Q, no longer anon: -q

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

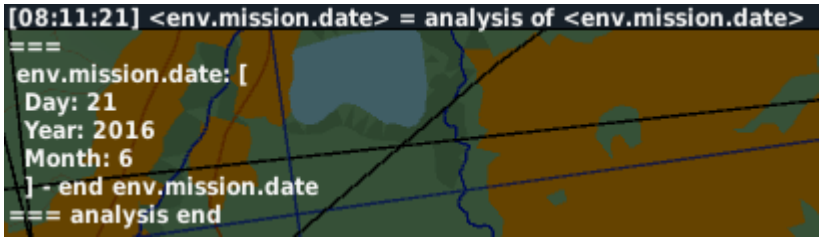




7.3.2 Analyze this: -a

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





7.3.3 W is for WARNING: -w



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

7.3.3.1 Live Bullets Here

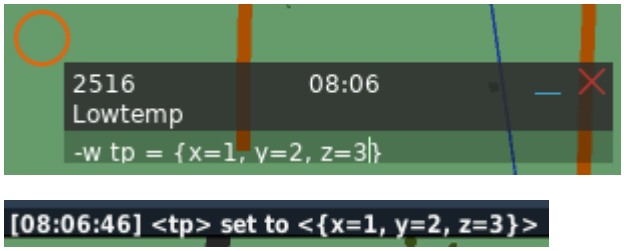
We aren't in Kansas anymore, Dory. The Debugger shoots with live bullets, and like Mission Editor, there's no Undo. If you change or overwrite a table, that's that. If you accidentally crash the mission in the process, that's often the *best case* scenario.

*7.3.3.2 Yes, **any** legal Lua Expression*

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

7.3.3.3 A Mild-Mannered Example

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



And then



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

7.4 Big Brother: Observers

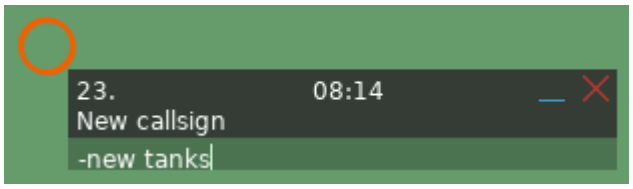
Most missions can be broken into separate, logical parts, and are designed accordingly. You may, for example, first design Blue's AAA, then ground defenses, then attack groups. Usually, and especially with DML, they use flags to communicate. The debugger supports this and allows you to group flags with 'observers' that even provide integration with Mission Editor to set up a debugging session before the mission runs.

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

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

7.4.1 Creating an observer: -new and new for

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

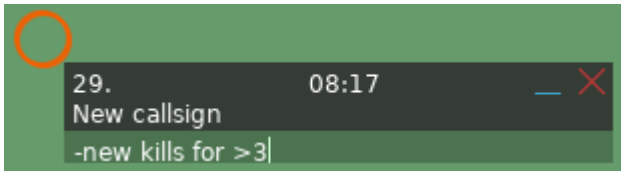


You can give observers any name you like, **but to guarantee compatibility with all commands, make sure it does not contain any blanks**. I prefer short and meaningful names, because you will have to re-type the name every time you add a flag to it, and because it's always displayed with the flag name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

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

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

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



We can change that: Instead of simply issuing '-new <name>' we can add a DML condition (also called 'Watchflag Method') through the optional 'for <condition> to the observer. All flags that are added to this

observer are checked against that condition.

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

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

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

7.4.2 Adding flags to observers: -observe with

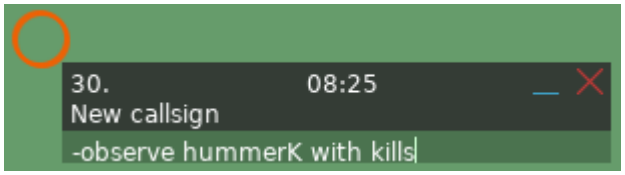
You add flags to an observer with the ‘-observe’ command.

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

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

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

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

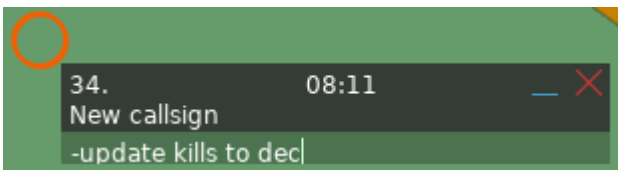



```
*** [08:02:28] debugger: now observing <hummerK> for value >3 with <kills>.
```

7.4.3 Changing an observer’s condition: -update to

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

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



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

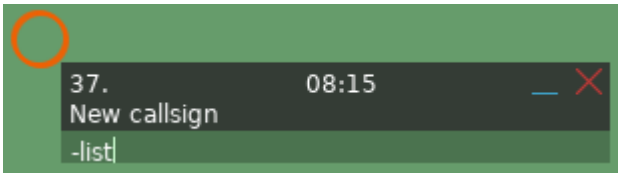
REMIONDER:
Remember that an observer name that contains blanks cannot have their condition changed. So if you repeatedly get an error whilst trying to change a condition, verify that the observer name is free of blanks.

7.4.4 Supported Observer Conditions

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

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

7.4.5 Show me what you got: -list



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

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

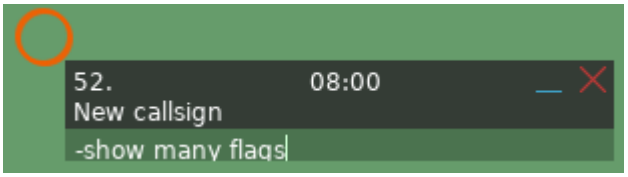
The list above shows a total of 5 active observers, the number of flags they each track, and what they are looking for in the flags they track. For example, the observer called "Look for pls = 15" triggers when a flag it observes changes its value to 15, and it is currently tracking 1 flag.

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

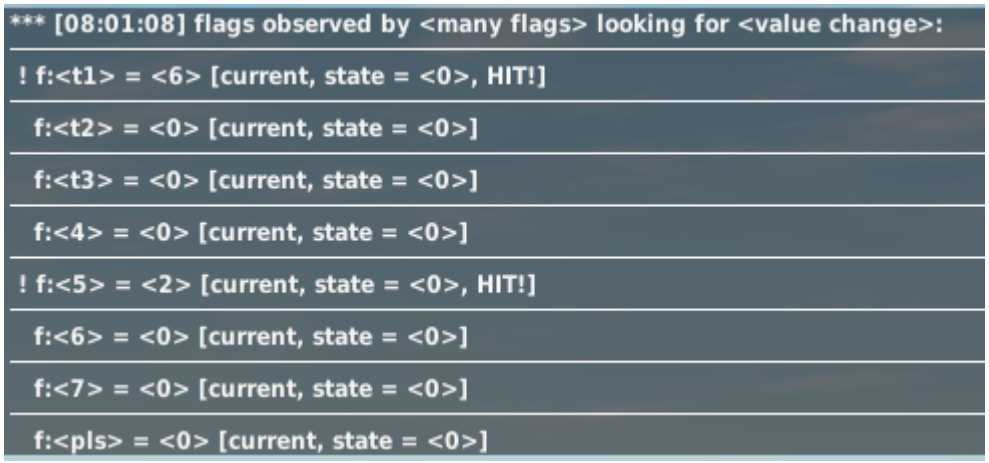
Also note that from the early mission time (25 seconds into the mission) it is obvious that we are looking at pre-set observers created in ME – more on that later.

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

7.4.6 Show and tell: -show observername

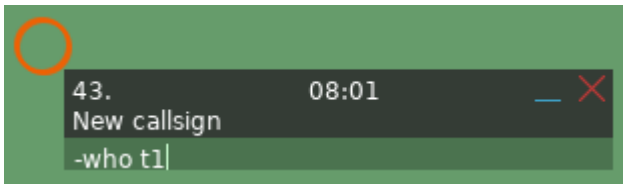


Just like you can tell the debugger to show you the value of a single flag, you can tell it to show you all the flag values that are managed by an observer. Simply use the observer’s name instead of a flag name and you’ll get a list of all flags that it is currently watching, along with their current values, and the condition that the observer looks for to alert you to a change:

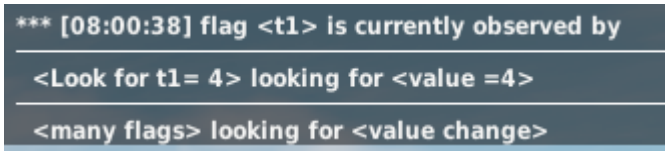


Note that the exclamation point in front and the text in square brackets (e.g. [current, state = <0>, HIT!] are only relevant (and can only occur) when you have disabled the debugger. When current and state values diverge (indicated by the exclamation point and ‘HIT!’ legend), the debugger will produce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

7.4.7 Who’s zoomin’ who: -who



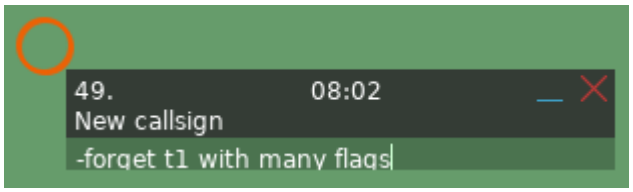
When your mission becomes complex, and you have multiple observers tracking your flags, it may become helfult to find out which observers track a particular flag. This is when you can employ the -who command, which runs through all observers to tell you which observer is checking out that flag, and what it is looking for:



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

7.4.8 … and Forget Me Nots: -forget with

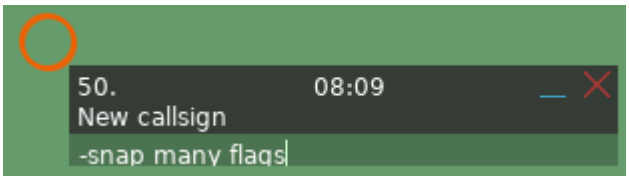
You can selectively ‘forget’ a flag from an observer by adding ‘with’ and the observer’s name to the ‘-forget’ command. This removes the flag from that observer list, and retains it on all other observers that also observe it.





7.4.9 Oh, snap! -snap observername

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





7.4.10 The spotless mind: -drop observername

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





7.4.11 Main switch: -start and -stop

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

7.5 Saving the Debugging Log: -save [filename]

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

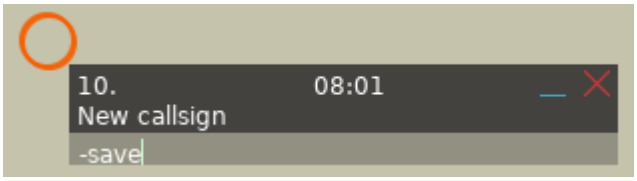
7.5.1 Using -save [filename]

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

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

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

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





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

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

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

7.5.2 How to enable -save

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

7.5.3 Do I have to add persistence?

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

7.6 Integration With Mission Editor (Optional)

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

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



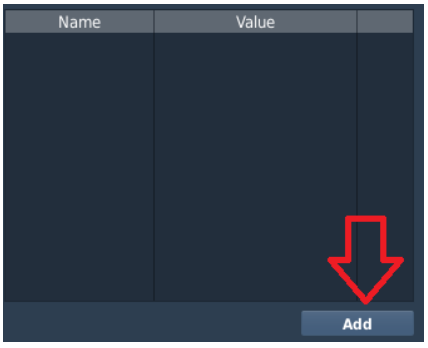
Of course, using this feature is optional and entirely up to you. As soon as you add the debugger to your mission, it's good to go: press 'Fly Mission' and you're off. If you merely intend to hunt down a single, particularly obstinate bug, that may be a viable approach: start debugging right away before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

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

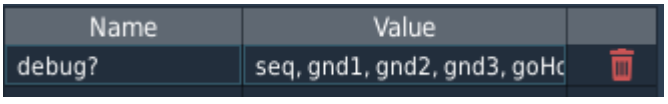
7.6.1 Creating Observers in ME

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

To create an observer in ME, simply



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



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

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



You can also set the observer’s trigger condition with the “debugTriggerMethod” (or one of its synonyms, like “sayWhen”) attribute. It defaults to ‘change’ but any DML Watchflag Method is valid. Please be advised that terms like “DML Watchflags” and “synonym” are DML parlance that will be inaccessible to you until you try your hand at DML. You can use the debugger without knowing that stuff, you merely can’t max out its usefulness like DML warriors can.

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

7.6.2 Dedicated and Stacked Zones, tracking local flags

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



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

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

7.6.3 Activating Event Monitoring

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

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

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

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

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

7.6.4 Setting up Generic Stand-ins

Name	Value	
inf	Soldier M4 GRG	

You can again use a trigger zone to set up which units The Debugger spawns. Create a trigger zone, and re-name it “debuggerSpawnTypes” (spelling and capitalization is significant). Then add attributes and name them after the generic type, and set the value to the DCS type name (see here https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB) that you want The Debugger to spawn instead. You only need to add attributes for those generics that you want to replace.



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

7.7 Adding The Debugger to your mission

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

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

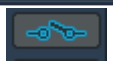
The stand-alone version is provided for those who have never ventured into the fun world of DML enhanced mission scripting, and simply need a debugger NOW!!!!one please. If you are already using DML, please use the DML-version. The stand-alone version may lack features of the DML version and is not maintained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

7.7.1 Stand-Alone (NO DML in your mission)

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

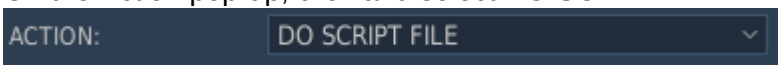
- Open the mission in ME
- Add a "trigger rule" to your mission:



  - Click on the set rules for trigger icon 
  - Under the ❶Triggers heading, click on the NEW button
  - From the new fields that show up, click on Type: 1 ONCE pop-up menu and change that to 4 MISSION START



  - On the far right under the ❷ ACTION Heading, click NEW
  - On the Action pop-up, click and select DO SCRIPT FILE



  - Click on the OPEN button and select "Mission Debugger.lua"
  - Save the mission.

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



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

7.7.2 DML-enhanced Missions

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

WARNING

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

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

7.7.3 ME Attributes

7.7.3.1 Debug Zones (for setting up Observers)

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

7.7.3.2 Event Tracking

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

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

7.7.3.3 Generics

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

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

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

	starts: `+++debug: changed generic 'inf' from <Soldier M4> to <Soldier M4 GRG>` Defaults to false, no logging of generic changes

All entries are optional

7.7.3.4 Debugger Configuration

To configure the debugger module via a configuration zone,

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

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

7.8 Bug Hunt - A Live demo

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

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

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

7.8.1 Please ignore me: Self Test

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

7.8.2 Observing and setting flags to debug your mission

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



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

- *goElvis*
  A flag that activates the Herc on the runway. It'll then take off after a brief interval

- *elvisLeaving*
  This flag turns true when the transport 'Elvis' leaves the 'Home Plate' zone around Batumi (All of group outside). Elvis is airborne at that point and turns towards

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

- *elvisSafe*
  This flag turns true when the transport arrives at Kobuleti. It signals a successful conclusion of this mission. Remaining attackers should turn home, and some congratulatory message should be displayed. We simply trigger a message in the demo

- *goAlba*
  A flag that activates the aggressor Albatross. This flag turns true when Elvis has been activated with goElvis and elvisLeaving is true

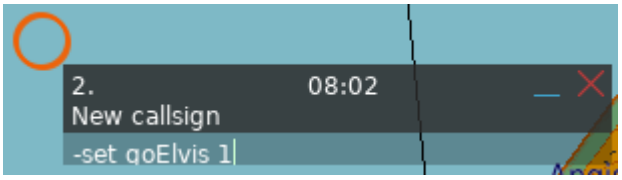
So, let's run the mission. Initially all seems well: We have a cool Radio Menu wired into flag 'goElvis' (provided by a DML radio menu zone), but we will be using the debugger just because we can.

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

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

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

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



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

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



Ok, how did that happen?

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

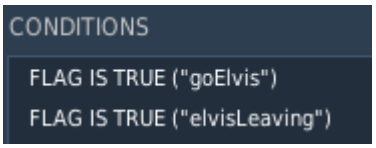


Now start Elvis with "-set goElvis 1"

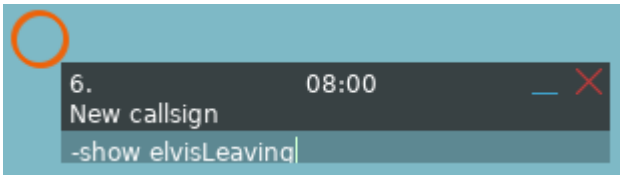
And… As soon as we set goElvis to 1, the debugger reports a change to goAlba: it goes from 0 to 1, triggering the Albatros to spawn.



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



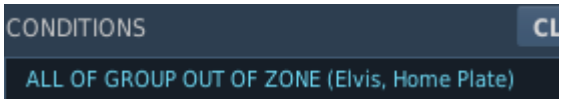
Let's have a look. Restart the mission, and "-observe goAlba" as before. But since we are now suspicious, let's also examine "elvisLeaving" before we spawn Elvis. We expect this value to be zero, since Elvis hasn't left yet.





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

In hindsight that's correct. elvisLeaving is defined as



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

7.8.3 Setting flags to skip/advance mission stages

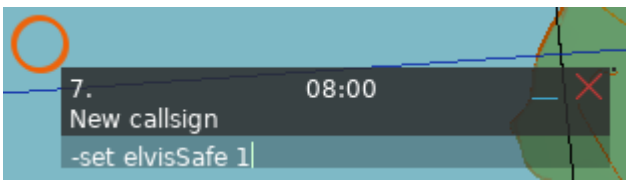
The next use case is so obvious that I only mention it for completeness:

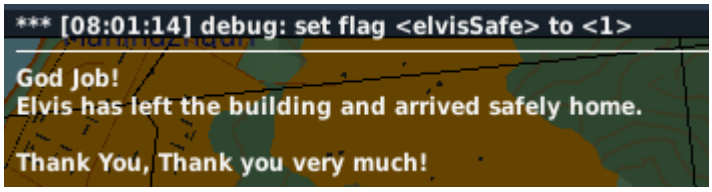
A common mission design pattern is to divide a complex mission into smaller, more manageable 'phases' like start-up, assemble, ingress, attack, egress, and recovery. For the mission to know which phase it is in (i.e., which action it needs to trigger), flags are used. Using a debugger to trigger these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

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

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

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





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

7.8.4 Using Observers and snapshots to debug your mission

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

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



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

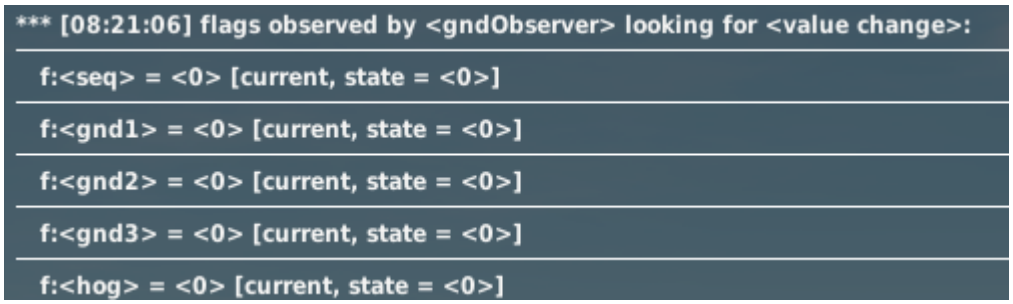
In our demo, the three groups Ground-1 through Ground-3 are controlled with flags ‘gnd1’, ‘gnd2’ and ‘gnd3’. We are also using a flag ‘hog’ to activate the hog [Note: we are using some *serious* DML magic in this example that reduces this task to little more than child’s play. See the notes at the end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

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

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

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

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



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

“-snap gndObserver”



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

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

“-set seq 1”

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

Then the Hog comes to live, and starts moving!

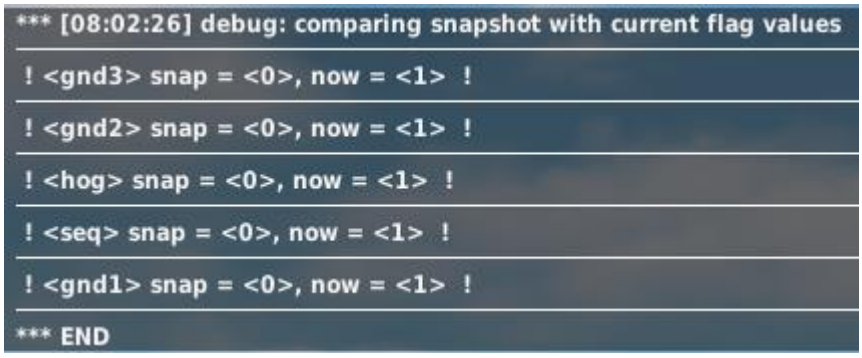
(you may also have noticed something strange: labels suddenly appear when units come to life – this is another DML feature that we ignore. Just marvel at the possibilities you have when you used DML)



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

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

"-compare"



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

7.8.5 Setting up Observers in ME

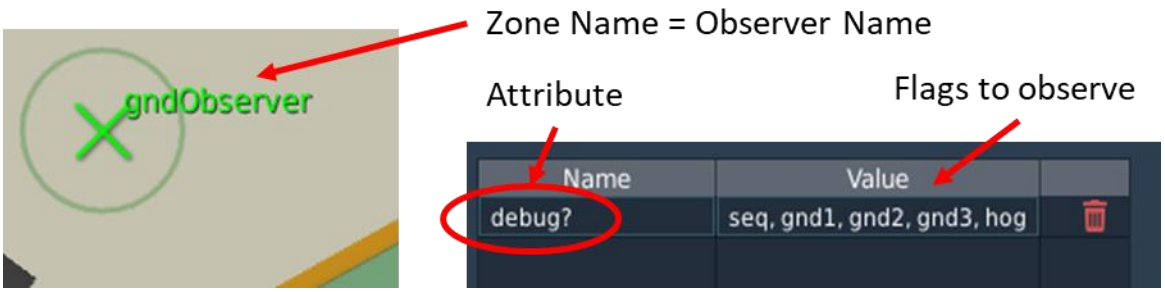
In the previous section, we manually built 'gndObserver', which we then used to track several flags and observed how they changed over time.

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

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

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

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



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

7.8.6 Debugging local flags, flag concurrency

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

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

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

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

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

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

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

So, let’s track the local *go” variable in the debugger. All we need to do is add a “debug?” attribute, with “*go” (*with* the asterisk) as value. Save the mission

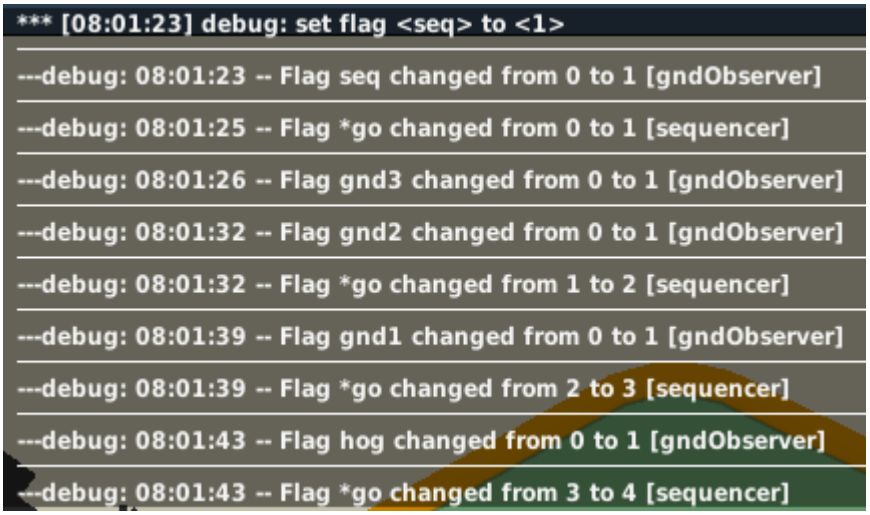
Name	Value
debug?	*go
pulse!	*go
pulses	4
time	4-7

and run it. The debugger loads the new observer and immediately activates it.

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

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

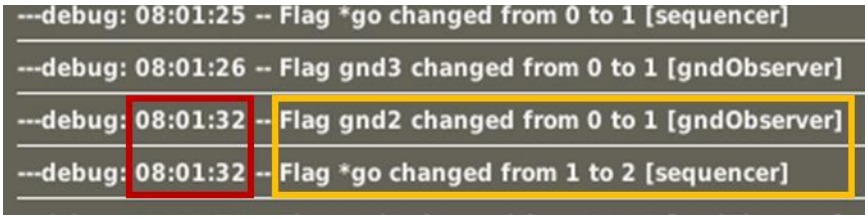
We also see that flag *go (the local) flag has gone from 0 to 1. The only way to know which zone this flag belongs to is by looking at the end of the line, and we see the observer's name (which matches the zone name) in square brackets: [sequencer]. So, we know that sequencer's local go flag went from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



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

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

The record, however, seems to indicate otherwise. In the amber box on the right, signal "gnd2" is listed before "*go", making it seem as if it happened before *go. It did not – This is merely a result of how the debugger works, and the time "resolution" it has. By default, the debugger samples all flags four times per second, and everything that happens in that same time slice it reports as concurrent – it does not know what came first, and simply reports changes in the order it walks through the flags. Since gndObserver's list of flags is worked down before starting on sequencer's, it reports gnd2's change before it reports *go's.



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

Note also that simply increasing the debuggers 'time resolution' (for example by slimming the time slice to 0.1 seconds, there is no guarantee that this can give better results: many DML modules run on the same clock, and therefore there's a good likelihood that the time interval between the pulser sending it's message and the randomizer responding to the

signal is less than 0.001 seconds. Keep the debugger's resolution at 0.25 seconds (or more) to keep a good balance between accuracy and performance.

7.8.7 Now hear this: debugMsg and sayWhen observer attributes

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

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

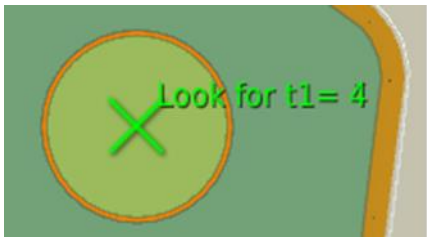
By default, the debugMsg attribute defaults to

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

which returns a message similar to

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

Look at the green "Look for t1=4" trigger zone



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

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

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

That is what triggers the output

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

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

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

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

7.8.8 Mission Discussion: What DML does in this demo

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

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

Impostors

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

Name	Value
impostor?	gimp
reanimate?	gnd1
onStart	yes

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

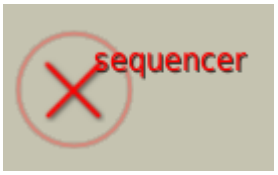
Name	Value
impostor?	imp
reanimate?	hog
onStart	yes
blink	0.1

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

The Vehicle Startup Sequence

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

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



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

The Pulser

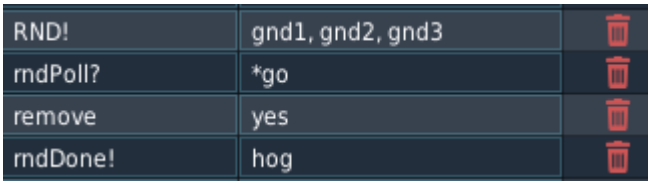
We want to start three vehicles, and the hog, making it a total of four (4) pulses that we want the pulser to produce for us. This is reflected by the 'pulses' attribute which switches it from the default 'infinite' to 4.
We also want a the time between individual activation to vary between 4 and 7 seconds, and that is what we find as value for the 'time' attribute (again replacing the

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

default 1 fixed interval). We also want the pulser to wait until it's started (onStart is set to false), and we send a signal to start on the startPulse? Input via the now very familiar 'seq' flag.

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

The Randomizer



We store the flags for all the ground vehicles in the RND! array, allowing the randomizer to pick one each time it receives a signal on the rndPoll?" input, which is wired to the Pulser via local "*go". We use the Randomizer's ability to randomly 'choose and discard' flags from its flags (remove = true), so that after sufficient poll requests have been received, it runs dry. We also use it's ability to signal that all flags have been 'used up' via the 'rndDone!' output. When this happens, we know that all vehicles have received their 'reanimate?' signal, and we can wire this signal directly into the Hog's Impostor reanimate? input, ensuring that it's always the Hog that starts last.

Shared flags and attributes

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



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

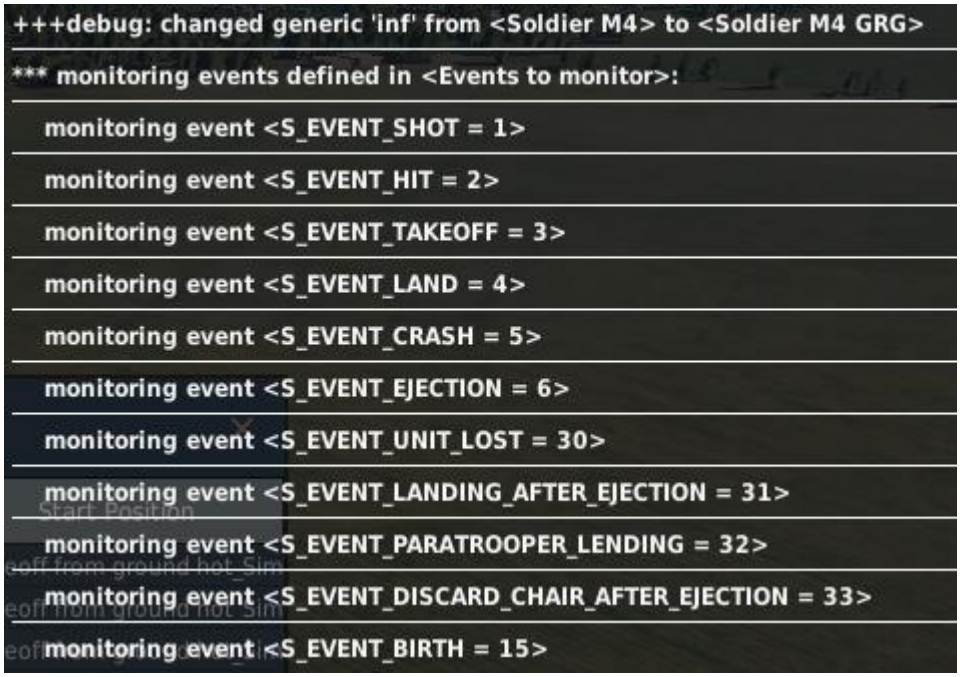
7.9 Debug events and More (.miz)

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

7.9.1 Starting the mission

Run the mission.

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



Take note, and enter “Frog One”.

7.9.2 In-Mission

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



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

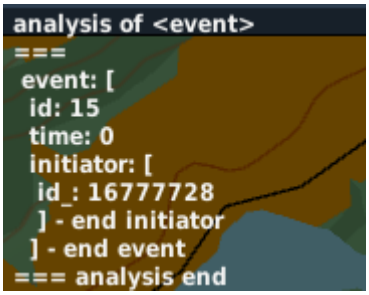
Since we are interested in this event (I want to find out what that event also tells us), go to F-10 Map view, activate the Mark Label tool, and click anywhere in the map. Invoke

```
-eventmon last
```



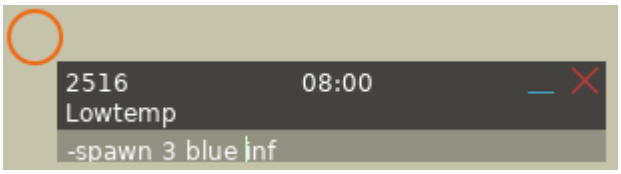
to get an analysis of the last event that The Debugger recorded for us:

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



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

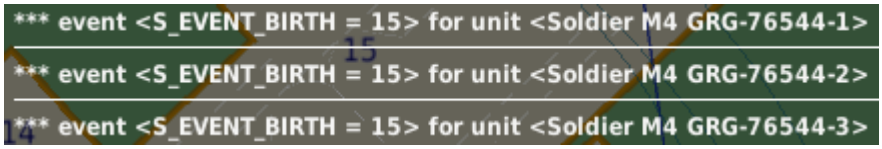
Next, we are going to try out The Debugger's "spawn" ability. Click on the Mark Label icon, and then place it close to and in front of your aircraft (for the simple reason that we can see what we spawn from the cockpit). Issue




```
-spawn 3 blue inf
```

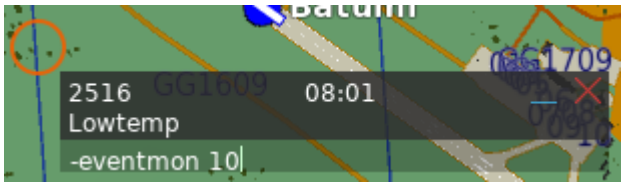
and click outside the text box to activate. If everything goes well, three new Soldiers (the Soldier M4 GRG type, the newer Soldier M4 kind with better animations) belonging to blue appear on the map close to the location where you placed the Mark Label.

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



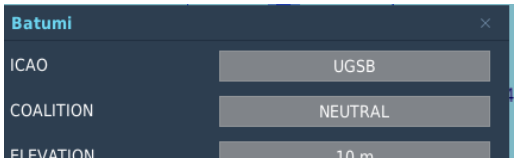
This of course corresponds to the three infantry units that we spawned. Since 15 (Birth event" is among the events that

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





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



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



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

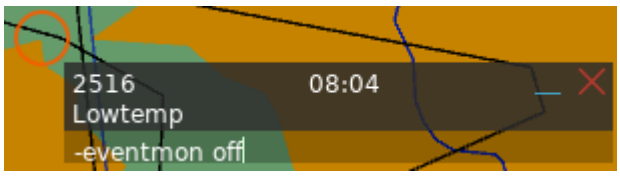


If we now click on Batumi again, we find that Batumi belongs to the RED faction.

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

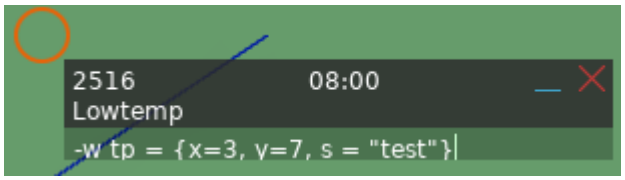
```
*** event <S_EVENT_LAND = 4> for unit <Lander>
```

Now clear all events from being reported.




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

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




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

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




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

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




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

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

PART VI:

TUTORIAL / DEMO MISSIONS

8 Tutorial / Demo missions

8.1 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.

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

- **xFlags – Field Day**
  Shows how to use the xFlags module to perform complex decisions based on multiple flags

- **Virgin (Civ) Air / Air Caucasus**
  Show-cases the CivAir civilian air module, and how it can be used to quickly populate

your map with non-threatening aircraft that you must tell apart from the bogeys.

- **Count Bases Blue**
  How to use the baseCaptured module, and then use to perform decisions based on the number of bases captured

- **Pilots At Their Limits**
  Put's the limitedAircrafts dynamic pilot limit through its paces. Lose too many pilots and your side loses the engagement

- **Gate And Switch**
  Introduces the Swiss-Army-Knife of modules: Changer. Shown are both modes: value conversion, and gated switch to prevent flag signals from flowing.

- **Good Grief (unGrief)**
  A showcase for the (mild) griefer-repellent. Sanction friendly kills when they happen more than once. Usually employed for MP PvE servers

- **The Danger Zone (PvP)**
  Using the unGrief module to only allow PVP in specially designated zones.

- **Reinforcements A La Carte**
  How to use the radioMenu module to order reinforcements, or even a mission restart.

- **Delicate Subjects**
  "Delicates" are units that explode if you so much as look threateningly at them. See here how to use them

- **Forever-looping Spawns**
  How to solve a common issue for (server-based) missions that have long (multi-day) uptime and need to remain fresh (i.e., groups get constantly refreshed)

- **Impossible Impostors**
  Impostors. The coolest module in the set. By far.

- **Being Persistent**
  How to save a mission to disk, and continue later

- **Sequencing Fun**
  Everything is a sequence. A quick demo that puts the sequencer to use for some nice start-up animations

- **Departures and Landings**
  How to use the LZ module to detect player taking off from airfields and returning.

- **Willie Nillie**
  A romp for artillery fans. The mission shows how you can easily integrate Willie Pete and Player Score to create a near-instant classic

- **Feats and AutoCSAR**
  A brief demonstration of the Feats ability for PlayerScore, and how you can add feats with CSAR. Oh, and autoCSAR is also new.

- **Slot Blocking and You**
  Multiplayer slot blocking made easy.

- **BFM Combat Trainer**
  A mission that shows off the new moving/lionked zone abilities – especially relative-turning a zone with a unit.

- **Formation Trainer**
  More moving zone fun, here combined with some advanced messaging fun

- **Hope you guess my name**
  Cloner's naming schemes put to some devilishly good fun

- **I say hello goodbye**
  The Valet – your friend at the airfield

- **Davy Jones' Rocker**
  Sub hunting for beginners

- **Taxi Police**
  The mission that I cannot stand behind

- **Big Score More Score**
  More Player Score fun – delayed scoring after landing

- **Effects with a flare**
  How to use the flare module for some unorthodox lighting

- **Players in the Zone**
  A short demonstration on the usefulness of PlayerZone

- **Not Too Shallow**
  How to remove those naval husks when the water isn't deep enough to take care of wrecks.

- **No Gap, No Glory**
  The ever-popular StopGap in DML.

- **No Gap, No Problem**
  StopGap's sibling NoGap in action

- **My First Factory**
  What separates an OwnedZone from a FactoryZone

- **Caucasus Hangar**
  Stop Gap show-off instamission

- **Sitting Ducks in a Barrel**
  The Siiting Ducks companion module to create strategically challenging multiplayer missions

- **Flag Score**
  Using the flagScore ability in PlayerScore

- **Take on TACAN**
  A quick TACAN demo

- **Civ Air International**
  Using departure and arrival zones with Civ air to simulate off-map flights

- **Airbase mine**
  How to control airfield ownership with the airfield module

- **My Immortal**
  How to use StopGap's "refresh" option.

- **Send in the Clones**
  The requestable attribute for cloners

- **All is what I own**
  Showing off the hierarchical ownership abilities of ownAll ownership management module

- **Bombs Away**
  The BombRange bombing trainer creator

- **Types and Civil Liveries**
  Demonstrating liveries for CivAir

- **Boom Boom**
  The groundExplosion explosion. Fun to watch.

- **Landing Lessons**
  How to use TDZ for your own training missions

- **PlayerScore to win**
  How to use PlayerScore to control win conditions

8.2 Smoke'em! DML Intro.miz

8.2.1 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.

8.2.2 What To Explore

8.2.2.1 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.

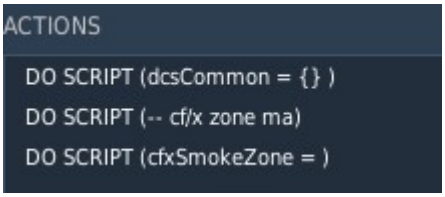


The smoke on the right, however, keeps happily on smoking. Oooh, rah, score one for DML! Yes, not that impressive, but let's move on to ME

8.2.2.2 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.

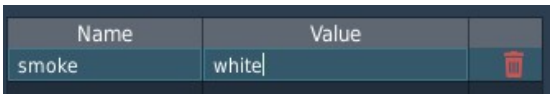
8.2.3 Discussion

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.

8.3 Object Destruct Detection (ME Integration).miz

8.3.1 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.

8.3.2 What To Explore

8.3.2.1 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 😊.

8.3.2.2 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



Name	Value
ROLE	
VALUE	
OBJECT ID	262537216
NAME	an-2m
f!	an2dead

Notice the "f!" attribute with a value of "an2dead". That is the only addition we made to the zone. It changes the value of flag "an2dead" by adding 1 to its current value when the object

named “an-2m” is destroyed. Since flag “an2dead” has a value of zero (0) at mission start, changing it to 1 (one) will trigger Mission Editor’s “FLAG IS MORE THAN 0” condition.

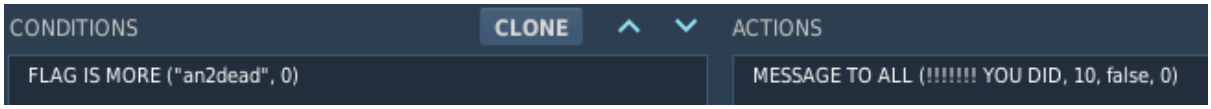
Now look at the ONCE trigger that runs when flag “an2dead” becomes greater than 0 (zero). That’s how you can integrate a DML module’s output with old-school ME triggers rules. Note that all DML modules provide integrated, much more convenient methods to detect and react to changes in flag values (DML modules internally and transparently use flags to send signals to each other by changing their values so you don’t have to set up trigger rules yourself). We merely go the old-school way here to ease your transition from classic/clumsy DCS trigger rules to DML’s greatly improved integrated way of mission design.

See “Demo – boom boom” for a fully DML integrated demo of the objectDestructDetector module that triggers three cloners after a building explodes.

8.3.3 Discussion

This mission uses the ObjectDestructDetector module to test if a map/scenery object was destroyed, and – when that has happened – the module reacts by changing a flag value. We use traditional ME-style triggers and rules to detect the flag change, and then put out a message.

To accomplish this, we simply added a single attribute to the ‘Assign as…” zone that ME created: the “f!” attribute, and set it’s value to “an2dead”, meaning that if the module detects that the object named “an-2m” (as was assigned by Mission editor) dies, it should change the contents (value) of the flag “an2dead” by adding the number one (1) to its current value. We then use standard ME trigger conditions to test if the flag named “an2dead” is greater than 0, and react accordingly by putting out a text message.



Of course, you can substitute that action with whatever you find appropriate in your mission.

Note that when you are using DML, using plain-vanilla DCS “Trigger rules” as we do here is the exception, since all modules have smart, advanced flag processing built in. You therefore usually simply “wire” module’s inputs and outputs together with a flag, and let the modules talk to each other. That way, you don’t have to maintain trigger rules in ME, the modules handle signaling and triggering transparently and without hassles for you.

8.4 ADF and NDB Fun.miz

8.4.1 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





8.4.2 What To Explore

8.4.2.1 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.

8.4.2.2 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.



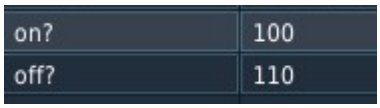
Note that although an NDB's frequency is defined in MHz, creating an NDB in

Name	Value
NDB	0.420
soundFile	distressbeacon.ogg
linkedUnit	Mover One

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

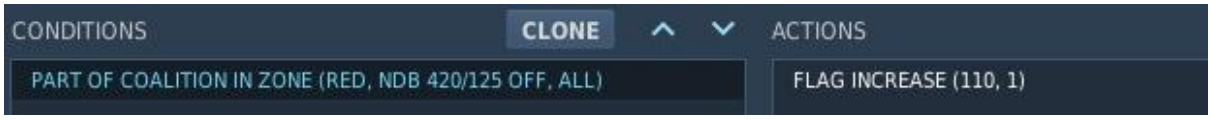


See how we use ME flags 100 and 110 to turn on multiple NDB at once by using the “on?” and “off?” attributes in NDB 420 KHz (the one following the Hummer) and NDB 125 MHz (the one in front of runway 24). Note that multiple NDB can share the same trigger flag.

8.4.3 Discussion

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

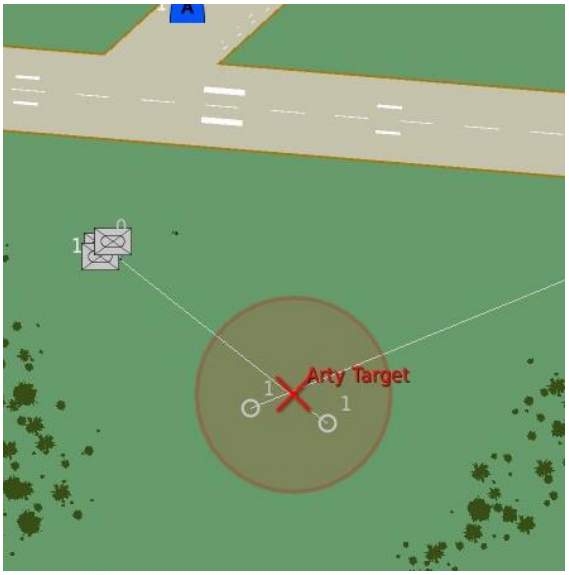
Being able to start and stop NDBs with triggers also allows some sneaky mission surprises, for example simulating ELT failure in the last moment when you are just about to discover the unit that you are looking for.



8.5 Artillery Zones Triggered.miz

8.5.1 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.

8.5.2 What To Explore

8.5.2.1 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"

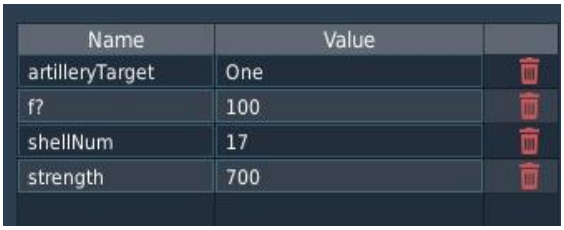
Notice that almost instantaneously, shells explode around those vehicles.

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

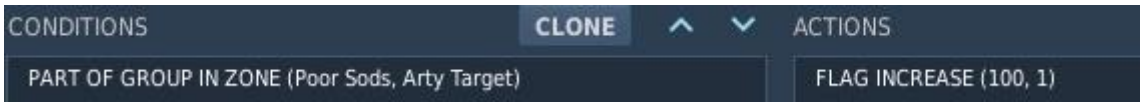
8.5.2.2 ME

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



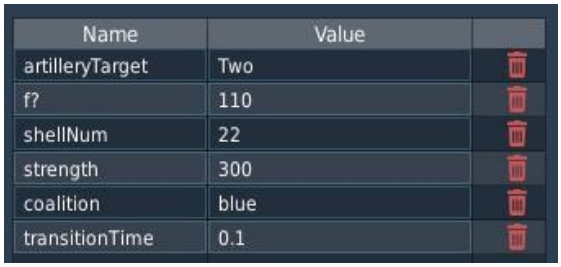
Name	Value
artilleryTarget	One
f?	100
shellNum	17
strength	700

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*.

8.5.3 Discussion

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.

8.6 ME Triggered Spawns.miz

8.6.1 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.

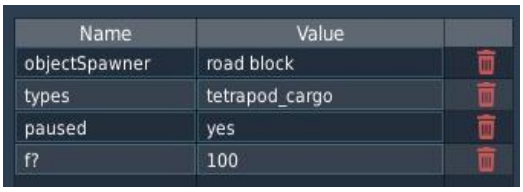
8.6.2 What To Explore

8.6.2.1 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

8.6.2.2 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.

8.6.3 Discussion

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

8.7 Spawn Zones (training and lasing).miz

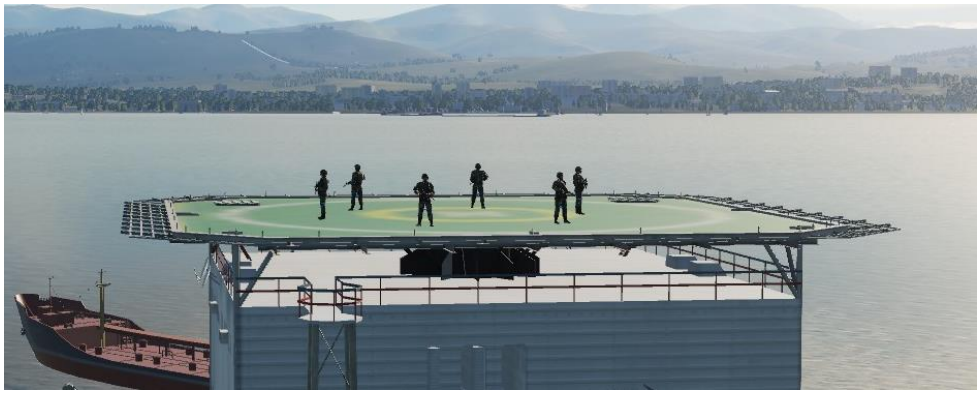
8.7.1 Demonstration Goals

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



8.7.2 What To Explore

8.7.2.1 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)

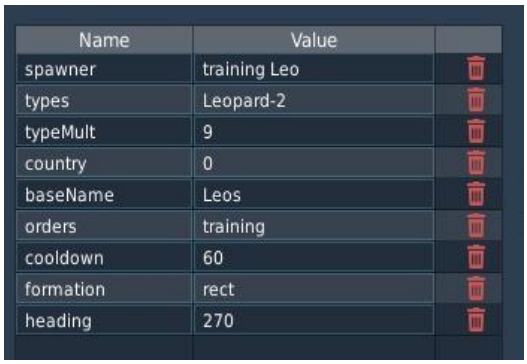
Destroy the red vehicles (you have unlimited ammo). After you destroy a group of ground vehicles, they will re-spawn (after a delay of 60 seconds). Respawns are unlimited

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.

8.7.2.2 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.

8.7.3 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.

8.8 Random Glory / Random Death (rndFlag)

8.8.1 Demonstration Goals

The module to kill for. Yes, *finally* a way to 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) I know that it will soon become one of your most-often used module.

In “Random Glory” we use the rndFlags module in a benign manner: to randomize a sequence of events: to select and activate a bunch of smoke zones and object spawn zones.

In your real missions, however, your actions are likely to be quite different: you’d be choosing and activating groups of enemies, all armed to the teeth and intent on killing the hapless players. See “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.

8.8.2 What To Explore

8.8.2.1 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

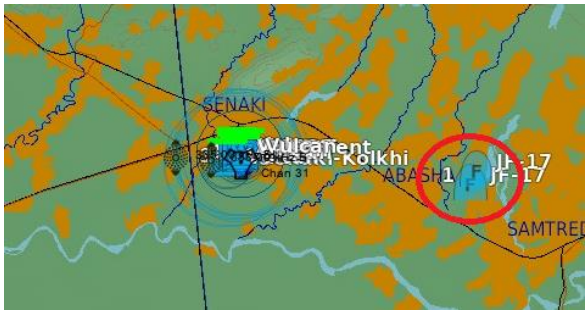
Start the mission in your Frogfoot. Look at the map. Your targets are the vehicles close to the tent on Senaki-Kolkhi's tarmac (marked with a red circle). Destroy them and you win.



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.

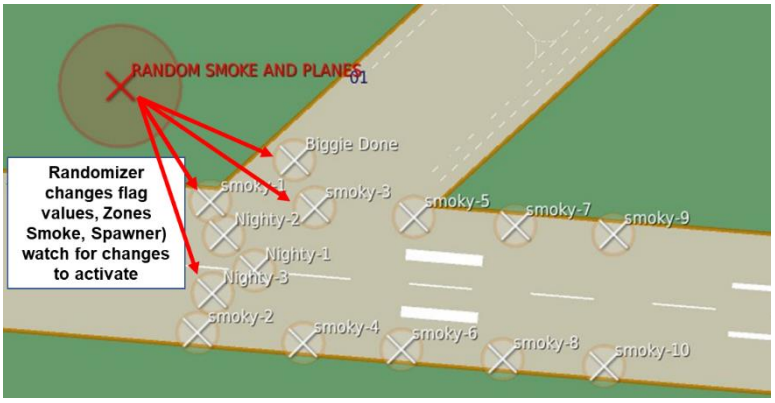
8.8.2.2 ME

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 connected to their input for changes (which triggers their action), or they can change flags that are connected to their output as a result of their action. Which means that if we connect an output of one module to the input using a flag as 'wire' to transport the information, we can eliminate ME's Triggers. And that's what we are doing here: the random generator creates signals on flag, and the flags connect to the input of smoke zones and spawn zones that then triggers their action.

In this mission we first set up a bunch of smoke and spawn zones, and key their inputs to different flags to activate. For example, Smoke Zone ‘smoky-1’ is keyed to activate when flag named “1” changes, while ‘smoky-2’ activates when flag named “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 the flags connected to their input to trigger a spawn. These spawners look at flags “A30”, “A31” and “A32” to trigger.

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

Finally, we added a randomizer that triggers each time that the value of the flag named “100” changes: f?=100. When it triggers, the randomizer

Name	Value
RND!	1-10, A30, A31, A32
pollSize	1-3
remove	yes
f?	100
done!	weAreDone

- picks one to three flags (pollSize 1-3)
- from of the set of 13 flags (1-10, A30, A31, A32)
- and sets them to 1 (it increases their value by one, which was set to zero when the mission started up. The ‘inc’ (increase) method is set by default)
- afterwards, the flags that were drawn, are discarded (remove = yes)
- When all flags have been drawn from the set, and a new cycle is requested on an empty set, the randomizer increases the value of flag “weAreDone” that is connected to the output “done!” by one (done! = weAreDone)

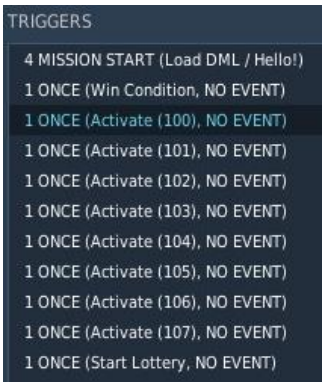
When you inspect the MISSION START trigger that loads DML, there are two additional actions that install the menu items “Change RND Watchflag” and that change the flag named “100” to the values 1 and 10, respectively. We use this setup because the randomizer looks for a change in the flag that is connected to its input “f?”, and menu items (unlike DML’s ‘radioMenu’ module) 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 by hande: we alternate between 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 then cycled once again, it sends a signal on the done! output (which is the flag named “weAreDone”) instead. That flag is connected to the input of 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 input watchflag.

Random Death

In contrast to ‘Random Glory’, this mission probably looks much more like a seasoned ‘plain vanilla’ Mission Creator 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. This is classic ME stuff.



Now comes the change: We take these triggers and feed them as a set of triggers into our randomizer and let it do the randomizing for us:

At some point in the mission, the Frogfoot leaves the starting zone. This triggers the ‘Start Lottery’ ME message, and at the same time changes the value of flag “99”. That flag connects to the input of the randomizer, which then picks a couple (3-5) of the flags listed in RND! at random, and sets them to one (1) -- randomizer actually increases each of their values by one, which, since at mission start they are all set to 0, will turn their values to 1. Since any flag value other than 0 is interpreted by DCS a ‘true’, it fulfils the trigger condition from the respective trigger rules, causing the units to become active and start hunting for you.

The randomizer’s setup itself also bears no surprises: It uses the set of flags as source set (100-107, a total of 8 items). It’s paused at mission begin (no “onStart” attribute given, so the default onStart value ‘false’ is taken), and waits for flag “99” that is connected to the input “f?” to change its value to start 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.

When that happens, the randomizer, triggered by a change in the flag “99” (f?=99) begins its work: 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.

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

Since method is implicitly set to ‘inc’ (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.

8.8.3 Discussion

‘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.

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

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. Basic 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** all delivered by a single, humble randomizer (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)

8.9 Pulsing Fun.miz (pulseFlags)

8.9.1 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.

8.9.2 What To Explore

8.9.2.1 In Mission

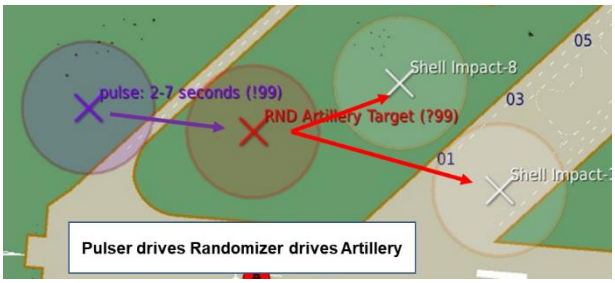
Enter the Frogfoot. Be sure not to move it an inch, or you’ll die. Open a beverage, and enjoy the show. Change to F2 outside view for an even more impressive display of destructive fun.



8.9.2.2 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

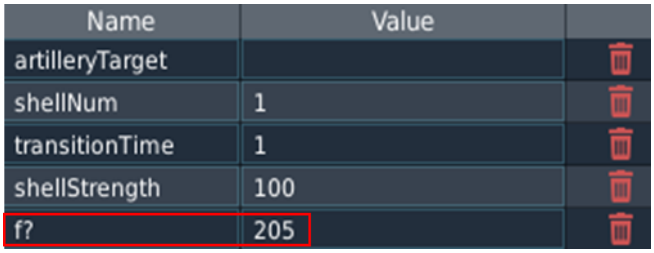
Th artillery zones all listen to their flags (200-210), and whenever its value changes, simulate a shell landing in their zone. This cascades nicely, completely unpredictable, into a great show. And no ME Triggers were hurt in this demo!

8.9.3 Discussion

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



Since DML inputs are looking for a value change in the flag connects to them, and we expect to trigger the artillery zones multiple times, we use the 'inc' method. This method increases the flag that is connected to the output by adding one (1) to its current value every time, generating a signal that triggers the input of the artillery zone modules. Note that "inc" is the default output method, and we could have omitted that attribute; I merely added it for more clarity.



We do this because the randomizer drives other DML modules (the artillery target zones all connect their input "f?" to flags named "200" through "210", and multiple artillery target zones connect their input to the same flag, meaning that they will activate simultaneously – an observer, however, will be hard pressed to tell which of these zones are using the same trigger), and their inputs usually react to a change in the value of the flag that connects to their "input?" ports.

Although this demo is a bit contrived (yet impressive!), it serves nicely to show 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.

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

8.10.1 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

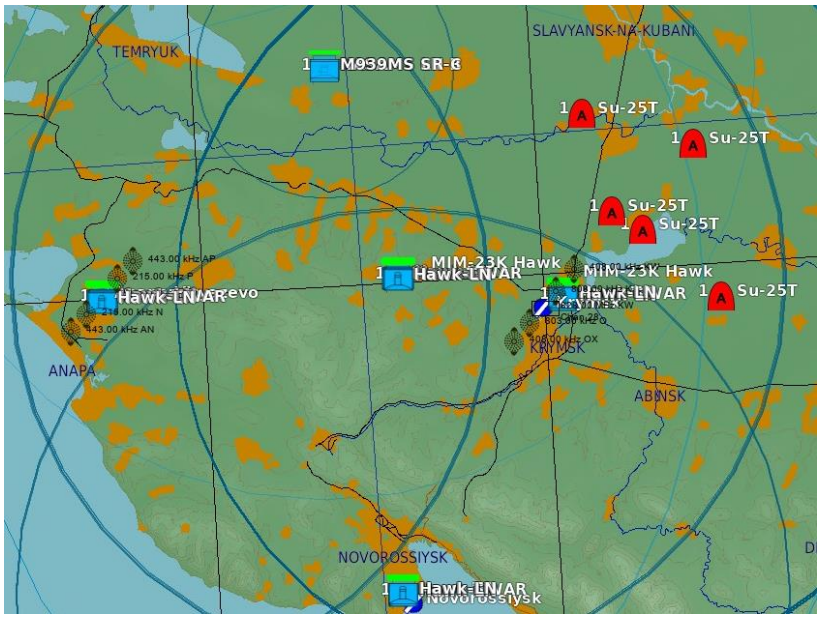
8.10.2 What To Explore

8.10.2.1 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.

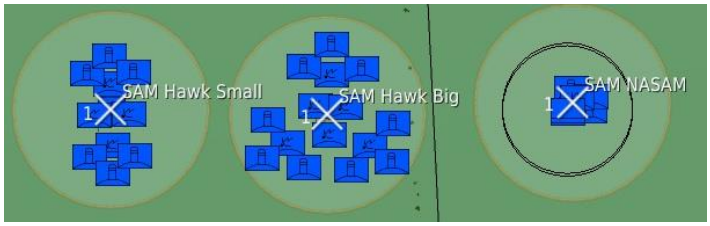


Nothing left to see, let's fire up ME and have a look at what's working behind the scenes.

8.10.2.2 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 random template (from the three it can choose from), and clones it. So each mission you have two to four

Name	Value
RND!	10-13
pollSize	2-4
onStart	yes

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, all groups that have at least one unit inside its zone are added tp the template, in this case the one SU-25T with orders to bomb Krymsk's runway will be cloned.

Name	Value
cloner	
empty!	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.

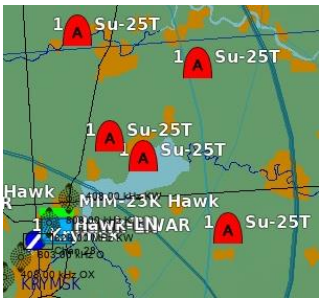
We also see with *spawn?* that every time flag 100 changes, this cloner will undergo a new spawn cycle. We also see that, strangely, *empty!* is also queued to flag 100, meaning that every time that all of the previously spawned clones are destroyed, flag 100 changes. This creates a feedback loop: when the last unit of the last cloned batch dies, a new spawn cycle is initiated. This continues until the last 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!	100
spawn?	100

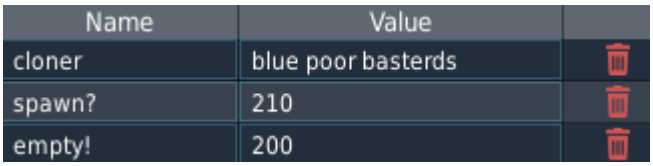
- They use Red Clone One as template, meaning that they spawn exact copies of whatever the template for Red Zone One is
- They do not have an *onStart = yes* attribute, meaning that when the mission starts, they don't spawn.

And there is something important that the cloners have in common, which we use for a surprise effect: they also *spawn?* and *empty!* 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

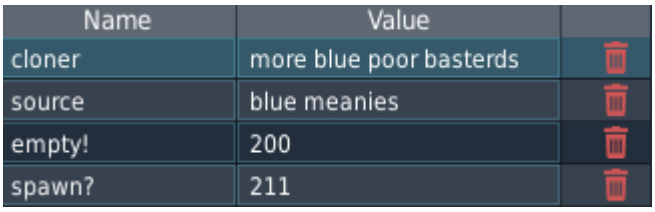
DELAYED ENDLESS RANDOM RESPAWNS



Name	Value
cloner	blue poor basterds
spawn?	210
empty!	200

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: 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!*.

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



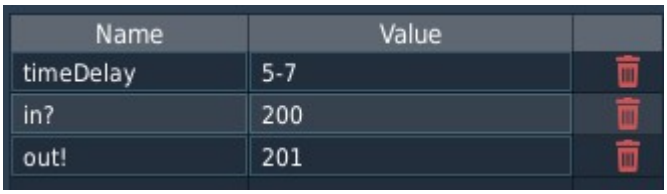
Name	Value
cloner	more blue poor basterds
source	blue meanies
empty!	200
spawn?	211

They have in common:

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

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)



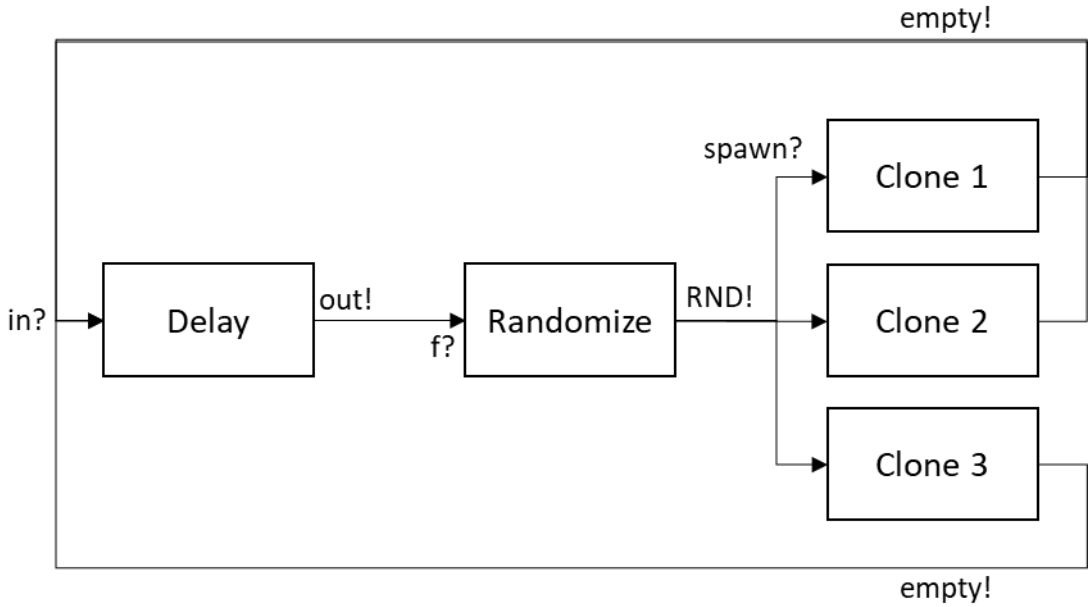
Name	Value
timeDelay	5-7
in?	200
out!	201

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!* 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
RND!	210-212
f?	201
method	inc

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 outputs *RND!* provides (which in turn each connects to one of the three blue meanie clone zones), completing the circle – a new group is spawned.



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:

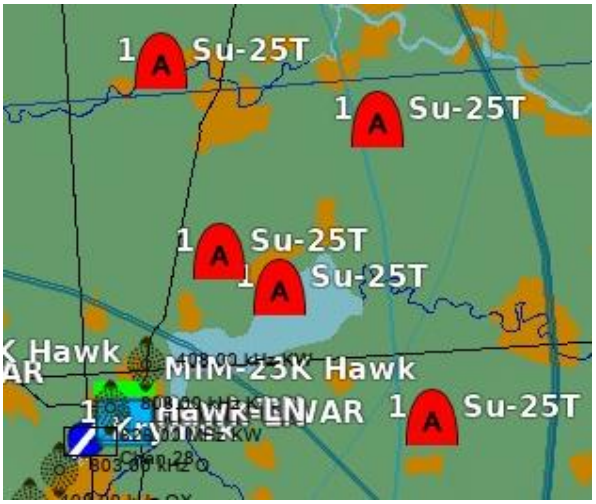
- This way, separate groups can have different routes even though they are spawned at the same time. Since routes are on a group level, you must use multiple groups if you want to assign different routes to parts of the spawn
- If you are using Helo Troops, it may be advantageous to split infantry and 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!* check, all static objects must also be destroyed if they are included in a template.

8.10.3 Discussion

"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!* attribute:

empty!	Number	When all units **from the last clone cycle** have been destroyed, this flag's value is changed according to method

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.

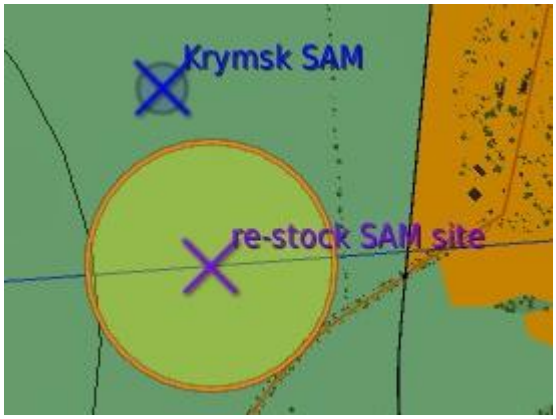
HOW DO THE Su25T FIND THEIR TARGETS?

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 uses 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 (onStart=no). If you change the 'onStart' attribute to 'true', it starts running, emitting a pulse every 5 minutes on flag 50. Flag 50 is the input for Krymsk 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.

8.11 Flag Fun – LOCAL FLAGS (& raiseFlags)

8.11.1 Demonstration Goals

Oh no, more colorful smoke! Well, sit down, and hold on to your shorts. 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.

8.11.2 What To Explore

8.11.2.1 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 color 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.

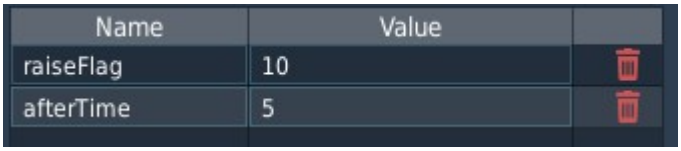
Fire up ME and load the mission. Secure jaw.

8.11.2.2 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:

SECTION OUTDATED, UNDER REVIEW

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:

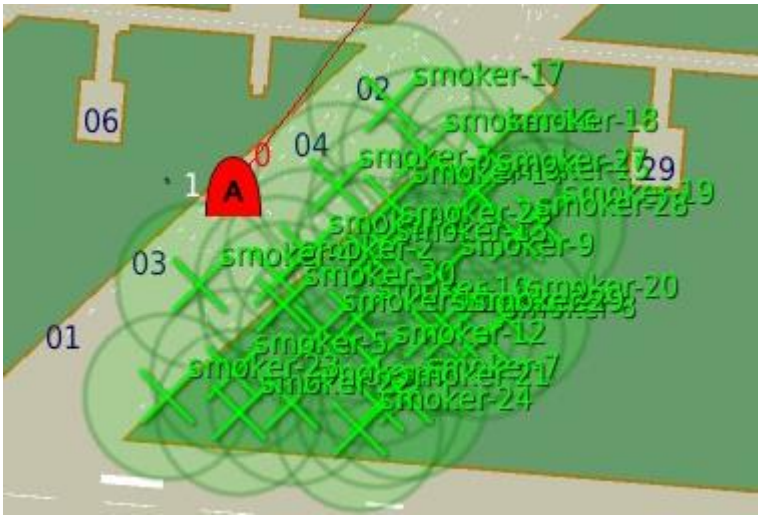
Name	Value	
smoke	random	
paused	yes	
startSmoke?	*smoke ON	
raiseFlag	*smoke ON	
afterTime	1-20	

- 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:

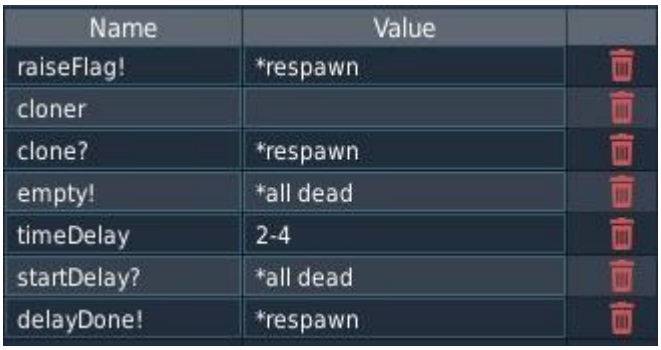
Inspect the other green “smoker” zones. Strangely, they are all alike – the only thing that is different for each zone is it’s name. In fact, it looks as the name was assigned by ME during lazy copy/paste. And that that’s exactly what happened. How can that be? Why doesn’t the first raiseFlag module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

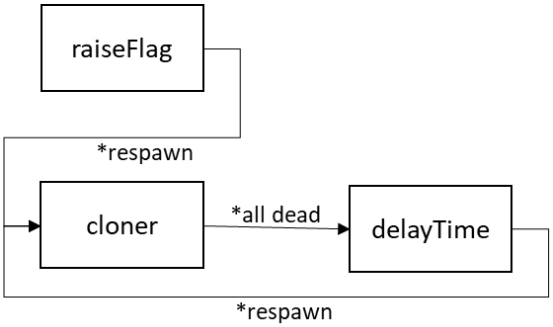


Name	Value	
raiseFlag!	*respawn	
cloner		
clone?	*respawn	
empty!	*all dead	
timeDelay	2-4	
startDelay?	*all dead	
delayDone!	*respawn	

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**!

8.11.3 Discussion

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.

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

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

This mission demonstrates how we can employ a counter to limit a cloner's spawning to a pre-set number. We do this the following way: we use flag to tell the cloner when to spawn. Once the determined number of spawns is reached, we no longer tell 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 use that signal from cloneZone to start a new clone cycle. We can do this with a simple feedback loop on the cloner from the *empty!* output to its *clone?* input. This creates an unending spawner as we have seen in *Attack of the CloneZ*, a tried and trusted design pattern.

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.

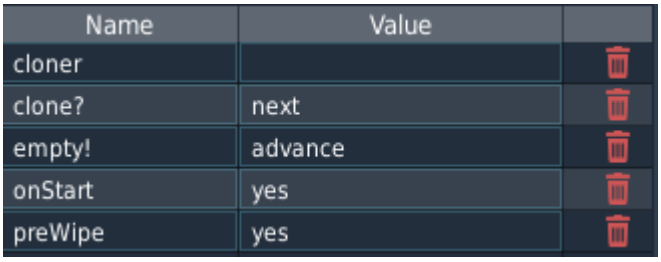
8.12.2 What To Explore

8.12.2.1 In Mission

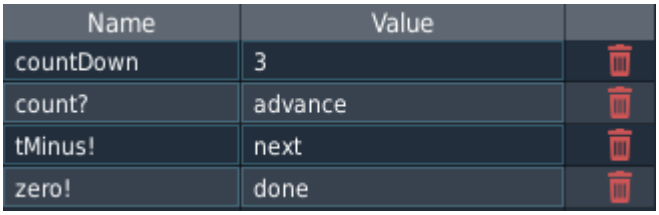
Jump into your trusty Su-25T. If you don't know how to use it to destroy ground units, invest the half hour in training, then come back 😊! - or use the Hog if you own Flaming Cliffs.

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.

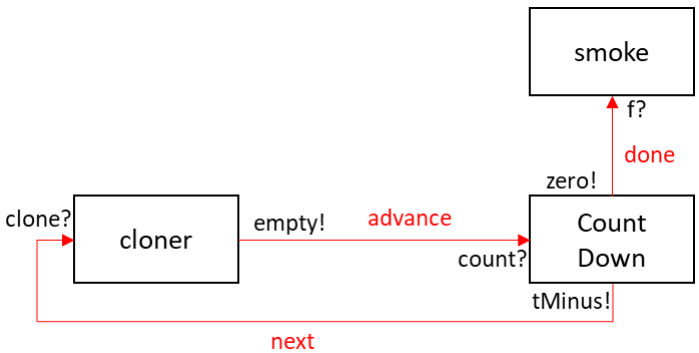
8.12.2.2 ME



There are few surprises here: we have a cloner that spawns the Hummer, and it is set up to spawn *onStart* (i.e., when the mission starts up), and every time that it detects a change on the flag that connects to input *clone?*, which is set to a flag named "next". Every time all units of the last spawn are destroyed (here the one Hummer), the clone module bangs the flag named "advance" (which connects to the cloner's output *empty!*). Since we need a way to stop the cloner from spawning at some point, we don't simply feed the output empty! back into clone? – that would create the well-known and useful never-ending spawner. Instead, we use another module to count the respawns and stop when we have spawned enough.



So, we feed "advance" into a count down module's input *count?* to count the *empty!* events, and use the count down's *tMinus!* output which propagates all input unless the counter has reached zero or less. We use the "next" flag to connect countDown's *tMinus!* output to the cloner's *clone?* input, completing the main part of this circuit. 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 value of 3 will result in 2 changes** on the *tMinus!* flag, which is exactly what we want).

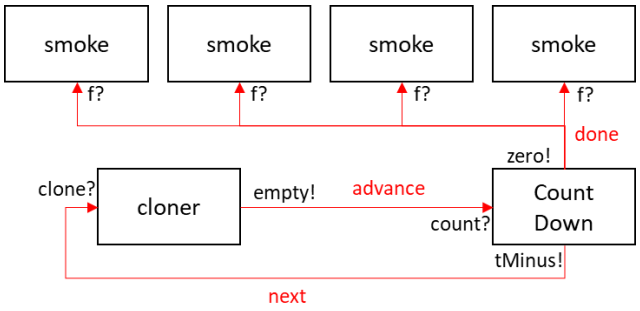


Understanding this is much simpler if you take a design approach like above: draw this out on paper, using boxes for modules like spawner, smoke and countDown, and draw lines between inputs with outputs. Then write the flag names that you want to use to connect to the in/output attributes over those lines. The result often looks like a digital circuit, and that is pretty much what we are designing here.



The last bit that completes the picture: we connect the smoke zone's *f?* (which starts the smoke) input to the counter's *zero!* output. This causes the smoke zone to start smoking when the counter's count reaches zero (here: after three kills). We use the flag that we name "done" to connect the in- and outputs, and we are done.

In the demo, we then use copy/paste to create four separate versions of the smoke zone and place them on the map around the intersection of the two runways. Since they are all configured identically, they are all waiting on the same flag named "done" to trigger their *f?* input. As a result, they all start smoking at the same time when the signal comes. The final circuit design looks like this:



8.12.3 Discussion

That was easy.

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 module.

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 😊

8.13 Bottled Messages.miz (Messenger, timeDelay)

8.13.1 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

8.13.2 What To Explore

8.13.2.1 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!

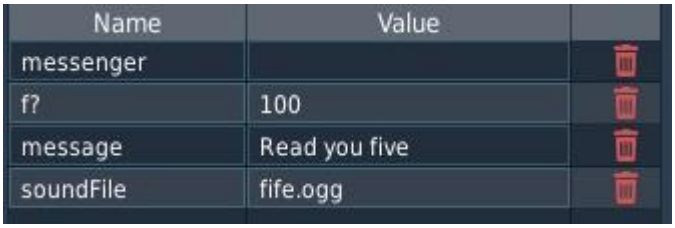
8.13.2.2 ME

In order to create user events, we use ME’s “Radio Item Add” actions that create a new entry in the Communications→Other tree.



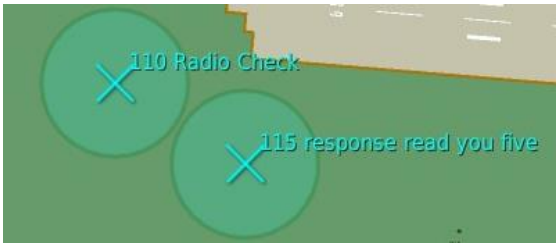
These radio items, when chosen, cause a flag (here 100 and 110) to be set to a specific value (note that ME can’t increase flags with radio items, just set them to a specific value, so any subsequent choice of the same radio item will not register as a flag change).

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

8.13.3 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.

8.14 Follow Me! (unit zones & messenger)

8.14.1 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.



8.14.2 What To Explore

8.14.2.1 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.

8.14.2.2 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

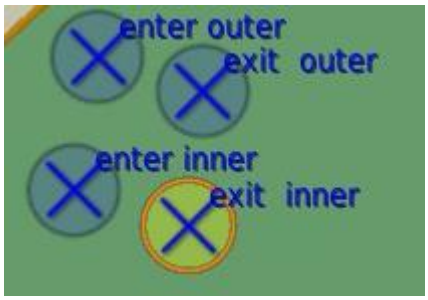
Stacked on the zone is also a messenger flag that activates when local flah ‘hi there’ changes, and that displays the greeting message that you saw when you entered the cockpit on the ground.

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:

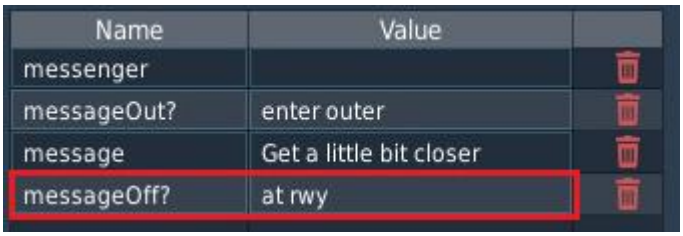


- “Get a little bit closer” when you enter the outer zone
- “That's close enough!” when you enter the inner zone
- “You could get a bit closer” when you exit the inner again and
- “Frog One, keep up!” when you leave the outer zone.

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?

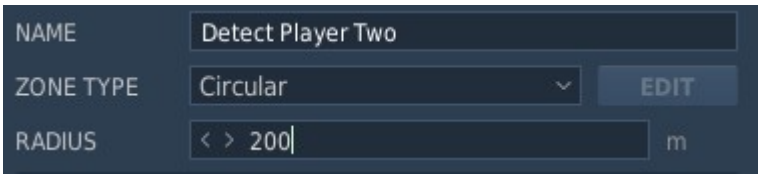
8.14.3 Discussion

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.

8.15 Clone Relations (Advanced Topic)

8.15.1 Demonstration Goals

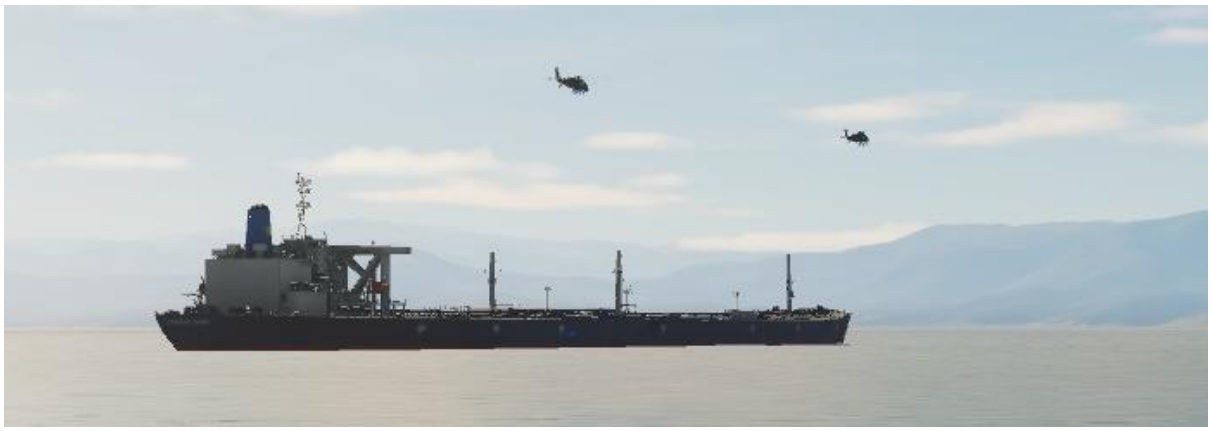
This demo explores the question: what happens to units/groups that are cloned and that have other units/groups in their waypoints?



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.

8.15.2 What To Explore

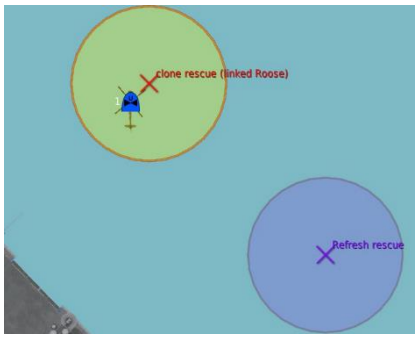
8.15.2.1 In Mission
This mission is a collection of micro-case-studies, and we want to explore them one at a time. Start the mission, then

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.

8.15.2.2 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



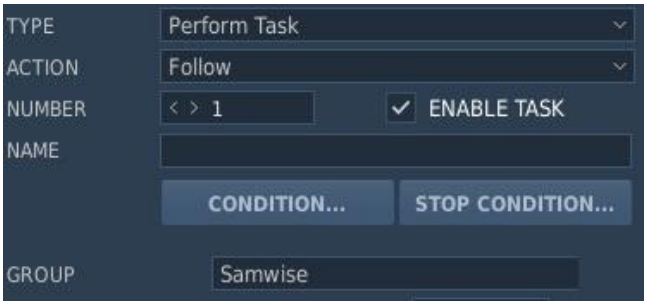
2. The Hornet is regularly spawned by the "Bug Cloner" clone zone. Note that since the Hornet is ser to "Takeoff from Runway" in its initial waypoint, we do not have to set the cloner to follow the Roosevelt via 'linkedUnit' (although it would not hurt to do so). Upon spawn, the waypoint reference

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.



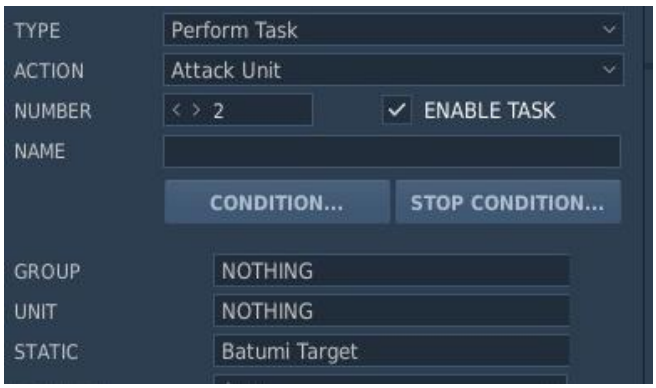
5. This example simply shows how clones (the falcons) can escort a normal unit, it's the purely



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)



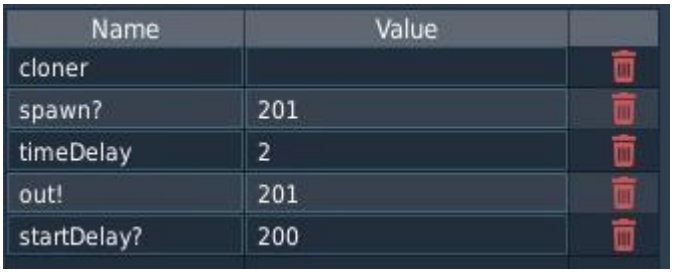
8. And finally, the tale of the endlessly battling tanks. Both groups have each other as targets defined, both are endless respawners (meaning their spawn is short-circuited with their empty! signal). The interesting question here is: how will this resolve, as theoretically, one will not have a correct 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.



8.15.3 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.

8.16 Moving Spawners I & II (SpawnZone and linkedUnit, GroupTracker)

8.16.1 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.



The “II” version of this mission also shows how you hand off spawned troops to group trackers

8.16.2 What To Explore

8.16.2.1 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.

Also note that in version II of this mission, we use a group tracker to track the number of troops (but not static objects) that are spawned and display a message containing the current count.

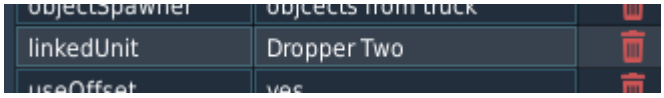
8.16.2.2 ME

Linking Zones

Note that there are multiple methods in DML to link a zone to a unit, and we use both in this mission. The simple (and newer) method to link a zone to a unit is to use the “LINK UNIT” interface element in the trigger zone’s interface:



Use this method when the unit to link to is accessible through ME. It's the preferred method and usually safer than the attribute based method we use to the other truck "Dropper Two" :



You should only use the latter, attribute-based unit link method in advanced cases where the unit that you want to link your zone to does not exist while you edit the mission but will be spawned later with a spawn zone.

Moving Spawners
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

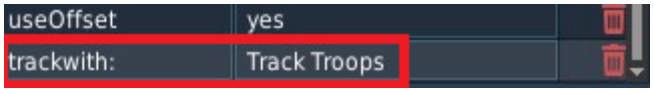
- **linkedUnit** is the second truck ("Dropper Two"), and **useOffset** is turned on. Since this an object spawner, we don't want the spawned objects to move with the linked truck, and therefore there is also an **autoLink = false** attribute
- **types** are two objects (Flag and Tire), and they are 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

8.16.3 Discussion

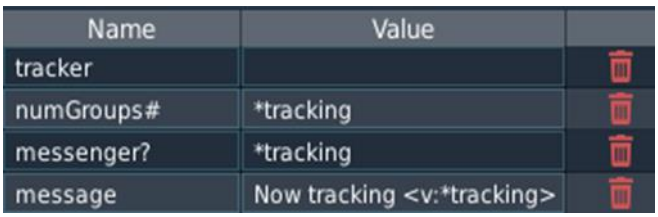
Even though it looks as if the trucks "drop" objects and personnel, in reality this is done with spawn zones that simply use their zone's linkedUnit attribute to follow a unit around.

Object spawners are often linked to moving, big units like ships, and are often set to spawn cargo/units onto that same object (carriers, assault ships), in which case the spawned objects need to be automatically linked to that same object as well. Since this is a common use case, object spawners default to linking their spawns to the same object that they are linked to. In our demo, we want the spawned objects to be 'dropped' to the ground, so we disable auto-linking.

Version II of this demo uses an identical setup, and additionally passes all groups (in our case single-unit infantry groups)



to a group tracker when they are spawned. Group Trackers can generate signals on their outputs when something happens to groups. To pass a group to a tracker after spawning, use the “trackWith:” attribute in the spawner.



When a new group is passed to the group tracker, it updates the number of groups currently tracked on the (local) “tracking” flag, which triggers a messenger module that is stacked on the same zone. The messenger then uses that same (local) “tracking” flag as number input to generate the “Now tracking XX groups” message that is displayed each time that the “tracking” flag changes.

Messenger uses the <v:*tracking> wildcard (see messenger documentation) to generate the real-time dynamic text that shows how many groups the tracker is tracking.

8.17 Helo Trooper.miz

8.17.1 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.

8.17.2 What To Explore

8.17.2.1 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 THAT YOU ARE LOOKING FOR!

Helo Troops provides its own menu in the "Other…" communication tree.

Choose communication→Other→Airlift Troops. This is the menu that connects to Helo Troops.

Note that the menu leads with Helo Troop's settings: Auto-Drop (currently ON) and Auto-Pick-Up (currently off). Select these to toggle their settings

Now for the more interesting stuff

Requesting A Spawn
You have the option to request spawning a group Legal Team Six. If you choose this option, the spawner connected to this menu item (automatically made by Helo Troops) causes that spawner to spawn a group and then undergo cool-down. Once the group is spawned, you can pick it up like any other group. Try again to spawn the group. If you are quick enough, you'll only get a message that the spawner is cooling down (well, you get a more appropriate message, but the cooldown is triggering this message).

Using spawners with requestable spawns versus immediate spawns is useful when you want a spawner to hold back spawning until the helicopter is very close, or the spawner sits in territory that can potentially be conquered (and spawned troops will immediately start fighting)



Loading Troops
Since Auto-Pick-Up is turned off, the helicopter didn't load the closest team, and you can choose which team to load. You have two options (three if you requested a spawn): Pick Me up, Pick Me Up Too (and Legal Team Six). Note that you do not have the option to load 'Illegal Team': they consist of infantry and an 'illegal' unit, the Hummer. Note also that the Team Missileer Pickup is also not available although it fully consists of 'legal' troops: it's too far away.

Choose one of the legal teams and load them up. They'll disappear from the game. Try to load another team. That's impossible, instead you have the option to disembark (deploy) the currently loaded team.

Deploying Troops (auto-deploy)
Now make sure that you have selected 'Auto Drop ON'. Take off, fly to the runway's center line close to the Su-25T, and land the helicopter on the runway. Your group of infantry disembarks immediately, deploying into a defensive circle around the helicopter.

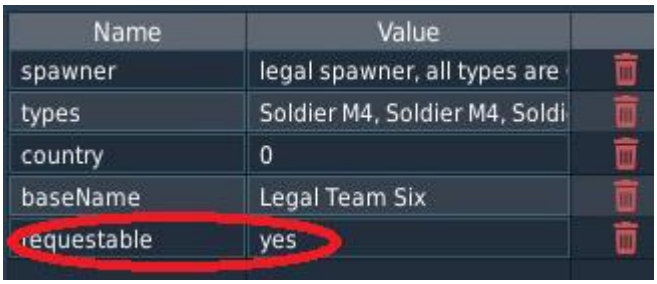
Auto-Pickup
Now make sure that auto-pickup is turned on. If you do this while still on the ground, note that the troops surrounding your helicopter are not immediately loaded – auto-load only happens the moment that your helicopter lands so you can safely change options while on the ground.

Take off empty, and touch down close to the Missileer group. That group is immediately loaded.

Manual Drop-Off
Now ensure that you have set Auto-Drop to OFF. With the missileer loaded into your helicopter, fly back to the runway where you unloaded the first group. Touch down and note that your infantry stays on board. Now choose Deploy Team to have your team of missileers disembark.

Weight Considerations
Currently, Helo Troops does not factor in the weight of troops it's loading into the helicopter. This is to be implemented later.

8.17.2.2 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.



8.17.3 Discussion
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!

8.18 Helo Cargo.miz – cargo spawn & receive

8.18.1 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 cargo objects, and the receiver zone uses ME flag “10” to count the number of objects delivered. We use old-school number-based flags names and ME trigger rules to output a message on the first, second and third delivery (as opposed to a much better solution using DML-based *messenger*). 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.

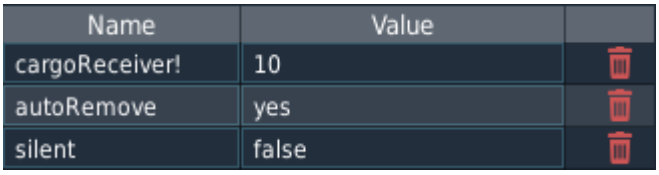
8.18.2 What To Explore

8.18.2.1 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.

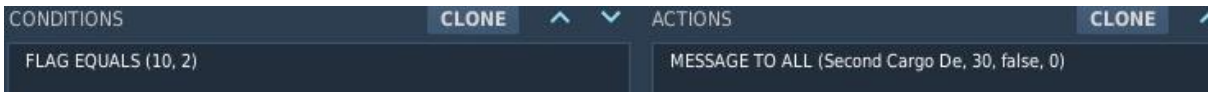
8.18.2.2 ME

Cargo is not placed with ME, **we use object spawners** to dynamically spawn the cargo. These spawners can supply an infinite amount of new cargo objects. The problem with these new cargo units: ME currently does not have the ability to detect when you deliver dynamically spawned cargo. A ‘vanilla’ mission can only work with cargo objects that you placed in ME before start of the mission. Enter DML: we have cargo receivers that can detect cargo delivery.



We are **not** using ME-supplied cargo trigger rules (e.g., CARGO UNHOOKED IN ZONE – because here they are woefully inadequate: they require that all cargo exist when the mission starts). Instead, the cargo delivery zone uses the **“cargoReceiver!”** output that is connected to the flag named **“10”** (an ‘old-school’ DCS number-named flag), which changes the flag’s value each time that a player successfully delivers cargo.

The messages are triggered by using standard ME trigger rules:



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.

8.18.3 Discussion

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 checks the location of the grounded cargo against all receiver zones. If that location is inside a receiver zone,

    - It sets the flags listed under the ‘cargoReceived!’ output according to the selected method. In our example, it increases the value of the flag named “10” by 1 (method ‘inc’, used by default).
    - 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

Challenges

- Instead of using ME-based trigger rules, use a *messenger* module and count all deliveries into the receiver zone.
- Instead of “10” use a better name for the flag/command

8.19 Artillery with UI.miz

8.19.1 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.

8.19.2 What To Explore

8.19.2.1 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.

8.19.2.2 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 changes the flag named "100" (Attribute **objectDestroyed! = 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

8.19.3 Discussion

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).

8.20 Missile Evasion (Guardian Angel).miz

8.20.1 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)

8.20.2 What To Explore

8.20.2.1 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

8.20.2.2 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.

IMPORTANT NOTE:
As of version 3.x (and later), GuardianAngel supports designating protection status of aircraft using zones and attributes. See “Guardian Angel Reloaded”, which explores the new features.

8.20.3 Discussion

Guardian Angel does its job really well, allowing missiles to come close, but not too close to protected planes. You can use this for many purposes: missile evasion school (where every time Guardian Angel intervenes, you would have lost), for adding harmless but blood pressure rising drama to a mission sequence (where a player plane receives protection to ensure nothing happens), or to kick up your missile training difficulty by disabling some Guardian Angel capabilities (for example disable interventions but keep missile warnings in place).

We have turned on the explosions effect in the configuration zone. Turn it off, and explore some other values.

What to try

- Turn off explosions
- Turn on ‘private’ – this reduces message clutter
- Turn off intervention and see how long you can survive. Mind the Missile missile missile! warning.

8.21 Guardian Angel Reloaded (Guardian Angel 3.x) tbc

8.21.1 Demonstration Goals

This mission builds on the previous ‘Missile Evasion’ mission, and adds details of the new features introduced with Guardian Angel 3.0 and later.



Guardian Angel adds the following capabilities:

- Dynamically turn Guardian Angel on and off during the mission
- Adding Aircraft to an ‘unprotected’ list using trigger zones
- Adding Aircraft to a ‘protected’ list using trigger zones

8.21.2 What To Explore

8.21.2.1 In Mission

Start the Mission in “GUARDED PLAYER” Frogfoot and as soon as you enter the cockpit, change to F10 Map view and zoom out. You’ll see a Falcon “Soon b ded” heading out to the east that, after a few seconds, is joined by another Falcon “Protected Falcon” heading the same way. Cycle through the F2 views until you are following “Soon b ded”. It won’t take long, and the first Falcon dies, so switch to “Protected Falcon” and follow that aircraft. This one should survive the barrage of three (!) S-300 SAM sites attacking it simultaneously (the S-300 is a SAM site that can re-target missiles in the air and beat the crap out of Guardian Angel 2.x). There is some residual risk that the protected falcon is killed by a missile that goes for a decoy that happens to be too close to it, but it usually survives.

Now, choose Communication→F10 Other→Guardian Angel→DISABLE Guardian Angel. It won't take long, and the Falcon is dead. Yes, S-300 are that lethal.

Turn Guardian angel back on using Communications→F10 Other→Guardian Angel and take off in your Frogfoot (you did choose the "GUARDED" slot, didn't you? Time to find out. Fly due east, at 2500m (some 7500 ft) and you'll see myriads of missiles being intercepted by GA.

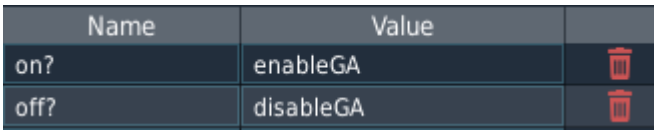
Now, using Communications, turn GA off. You'll don't have the ghost of a chance.

Turn GA on again and depart Senaki-Kolkhi in the "UNPROTECTED PLAYER" Frogfoot. You'll find that even with GA turned on, you won't get far. This aircraft, even though it's a player aircraft and GA is on, is unprotected, and the S-300 are far too deadly.

8.21.2.2 ME

In addition to being able to handle missiles that dynamically re-target (which the previous version of GA did not detect to the rude surprise of many players) version 3.0 mainly adds the following abilities

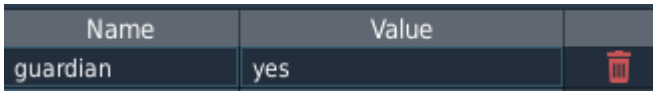
Turning Guardian Angel On and OFF



You now can use the config zone to tell GA which flags to watch for a change to enable and disable GA. When disables, GA will still track all missiles in the air (with negligible performance cost), but refrain from using that information to protect aircraft. When you switch GA back on, it resumes protecting aircraft, but be aware that it can be too late for some protected aircraft, as inbound missiles may already be in lethal range.

You use the two inputs on? and off? in the config zone to connect the Watchflags. In the demo we simply connect them to the radioMenu module that provides two items A and B.

Selectively Protecting and Exposing aircraft



You can selectively add and remove protection from aircraft by placing them in a trigger zone with the 'guardian' attribute. If the value of that attribute is false, the planes inside that zone will receive no protection whatsoever. If the value is true (or empty), all planes inside will always receive protection. Of course, aircrafts are only protected as long as GA is active (on).

Note that protection status is determined when the aircraft appears in the game. Moving through a 'guardian' zone does not affect an aircraft's status.

8.21.3 Discussion

So let's have some fun. Turning GA on and off does not necessarily have to be something you leave to the user. In many cases you may want a short time in your missions where you need to insure that players are safe (in an intermission etc), and for that interval you'd turn on GA. So, devise a way to turn on GA after 3 minutes, and turn it off after 5.

8.22 Recon Mode.miz

8.22.1 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.

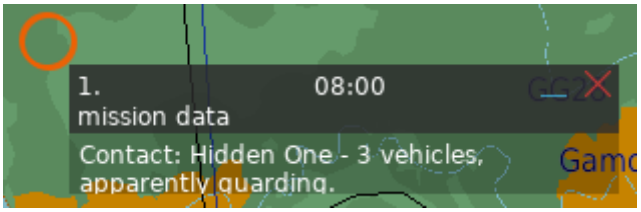
NOTE:
This mission merely demonstrates the bare necessities to get ReconMode up and run; it was originally written for ReconMode 1.x – the 2.x (or later) version included with DML adds some QoL features for adding and removing planes and ground forces to priority and other lists. The follow-up demo “Recon Mode – reloaded” explores these additions while building on the things learned here.

8.22.2 What To Explore

8.22.2.1 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-land 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.

Also note the text messages that appear on the right side of your screen. They appear whenever a group of ground forces is detected the first time, and they conform to the following format:

<*who*> reports new ground contact <*group name*>:
<*composition*>, <*doing what*>, <*location*>, at <*time*>:



<composition> currently describes how many vehicles, and how many infantry.

NOTE:
The report format has changed from 1.x of Recon Mode. This documentation reflects the new (superior) report format. You are welcome 😊

After a minute or so, a “GOTCHA” message appears. This is a **ME-triggered** message that is displayed after Recon Mode found a group that is on a priority list of targets

Be mindful of the fact that you never see a message that refers to the group of ground forces that is named “never find me”. This is important because this is a group that does exist, is squarely in the path of the recon plane, and was black-listed; 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 it sees.

8.22.2.2 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 group 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 ("detect!"): 100 (for normal discoveries) and 110 (for priority target discovery, "prio!")
- 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).
- Recon Mode supports adding groups to Prio- and Blacklist with zones. Inspect the next tutorial "Recon Mode – reloaded" to see how you can easily add priority groups and scout aircraft to Recon Mode

- 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

8.22.3 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.

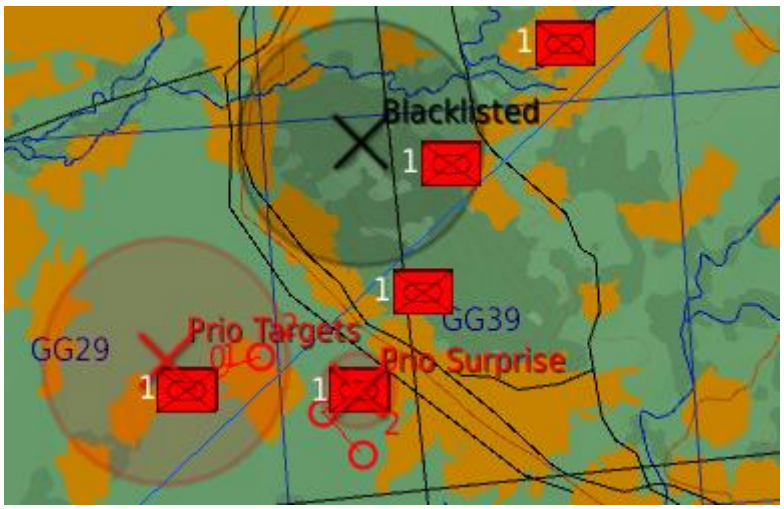
8.23 Recon Mode – reloaded (Recon Mode 2.x)

8.23.1 Demonstration Goals

This mission builds on the previous mission “Recon Mode” and focuses on the new features that came with 2.x of Recon mode:

- Selective inclusion / exclusion of aircraft with trigger zones
- Trigger-Zone based Priority- and Blacklisting
- Trigger-Zone based flags when groups are discovered (only priority targets)
- Better Reports
- UseRecon Mode’s detections to trigger game events
- Custom priority messages per priority zone
- Support for cloned groups
- Ability to individually suppress reports and map marks so detection events can be triggered without generating a report
- Turn Recon Mode on and off via flags

All 2.x features are in addition to 1.x and the module is backwards-compatible to 1.x



8.23.2 What To Explore

8.23.2.1 In Mission

Enter your Frogfoot ‘Gnd Player’, sitting on the ground at Senaki-Kolkhi, and switch to the F10 map view. Zoom out until you can see the Tomcat coming in towards the coast from the west.

As in the previous version, Recon sends reports whenever an aircraft detects ground units. The 2.x format contains more and better information, though: Let’s break it down:



Each report is now formatted as follows:

- *Who:* scout aircraft that is reporting
- *Strength:* number and type of units (infantry / vehicles)
- *Action:* if they are stationary or in which direction they are moving
- *Location:* in Lat/Lon (default) or MGRS
- *Time:* when they were observed

A heartbeat later, and as in the demo before, a 'Gotcha! priority group detected' message comes up.

A few seconds later, another – new - priority message appears:



Simply take note of the message contents.

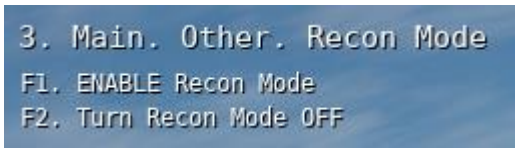
While the F-14 progresses further inland, turn your attention eastwards, where an F-16 has departed from Kutaisi, due East. It will overfly a group of red ground units. Note that there is *no* report. The F-16 *does not call in*.

Turn back to the tomcat (and probably Albatross which also will have taken flight by now), and after a short while you'll again see the message that 6 contacts have been spotted.

Now, change slots, and jump into the 'Blind Frog' at Kutaisi. Take off, and also overfly the enemy group to the East. Note that again, even though you are a player plane, no report is made.



Finally, change slots again into the "Air Trap" Frogfoot. A few seconds after you enter the cockpit, it detects some ground forces. Almost immediately you'll get a warning message 'Oh snap, it's a trap!': four new – previously unseen – groups have spring into life, and they are all targeting you. You can try, but you are hopelessly outmatched and will most likely be shot down within seconds.



Now restart the mission. As soon as the Tomcat reports the first contact, switch to Communication→F10 Other→Recon Mode and choose "Turn Recon Mode OFF". Now wait for a minute or two. You will not be surprised to see that no further groups are detected. Switch Recon mode back on again, and see that reports are starting to flow again.

8.23.2.2 ME

So let's go through the new additions to Recon Mode and what new tools you now have at your disposal:

Trigger Zones for Priority and Black List

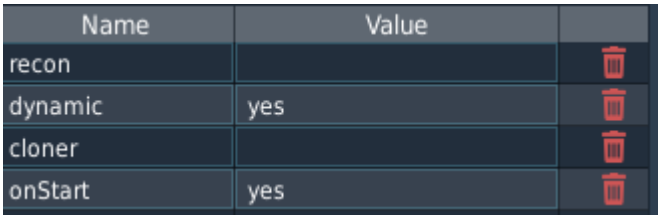


You now can use trigger zones to designate any group of ground forces that has at least one unit inside a zone with 'recon' attribute to be either a priority target, or blacklisted. As long as the value of the recon attribute does not start with 'black', the groups found inside are deemed to be priority targets. To blacklist them (recon will never discover them), simply enter the word 'black' as

value. Once units move outside of these trigger zones, they remain priority targets ow blacklisted.

Support for cloned groups
It's often desirable to have units spawn with cloners, and then be either blacklisted of assigned priority status. Recon Mode's recon zone supports this. Simply add the dynamic attribute, and all cloned groups that are based on the same naming scheme are automatically added to the corresponding list.

The demo mission demonstrates this by stacking a cloner on top of the 'recon' zone, thus ensuring that the group that is detected is actually a clone, not the original group (the onStart attribute ensures that a clone is created at mission start)



Priority Message
A new feature of Recon Mode is that you can add priority messages and flags to groups that are designated as priority targets with a trigger zone. Even if these groups move out of their designating zones, each group triggers an optional priority message and sets a priority flag when it is spotted.



The priority message supports a formatting very similar to the messenger module, with the exception that the location reported is that of one of the units of the group, not the center of the zone:

- <n> creates a new line
- <z> is replaced with zone's name
- <t> is current time in HH:MM:SS format
- <lat> the latitude of the discovered group's current position
- <lon> the longitude of the discovered group's current position
- <mgrs> the discovered group's position in MGRS coordinates

Important Note:
A priority message, when defined, will display even if the zone's silent flag is set to true. This is so you can override placing the map marker and scout report for these groups, and still be able to output a custom prioMessage.

Priority Flags

Just like a priority message, the defining priority zone can trigger flags each time one of the groups is spotted. As the “Oh snap, it’s a trap” example shows, this can be used in sneaky ways to initiate action (by using triggers) when specially designated units are discovered.

Name	Value	
recon		
spotted!	springTrap	

Priority flags are set even if the zone’s ‘silent’ attribute is set.

Staying Silent

You can suppress both recon report and map marks for any groups inside a priority zone by adding a <silent : true> attribute to a priority zone (does not work with blacklisted groups, as they never generate a report nor map mark).

Name	Value	
recon		
silent	yes	

Suppressing the report and mark can be desirable if you wish to use recon mode simply to trigger an event in your mission and still want to use normal reporting (i.e. keep the config’s ‘announcer attribute set to true).

Note:
Even when silent is set for a zone, output flags are set normally (when defined) and prioMessages are displayed (when defined) when a group is detected

Turning Recon Mode On and Off via Flags

The config zone allows you to specify two flags which Recon Mode monitors to turn on or off at will. We are using a RadioMenu module to provide the flag signals conveniently by menu choice, and set up the config zone to trigger on the DML Watchflags “turnOn” and “turnOff”.

Name	Value	
radioMenu	Recon Mode	
itemA	ENABLE Recon Mode	
A!	turnOn	
itemB	Turn Recon Mode OFF	
B!	turnOff	

Adding / removing aircraft from the scout roster

Recon Mode can now use Trigger zones to add and remove aircraft from the roster of active scouts. This always works on top of the settings chosen in the config zone. In the demo mission, Recon Mode is set up to assign the following aircraft as scouts:

- autoRecon (true): all aircraft automatically are added to the recon roster when then enter the game
- redScouts (false, via default): no automatic red scouts
- blueScouts (**true**, via default): all blue aircraft are automatically scouts
- greyScouts (false, via default): no automatic grey scouts

Name	Value	
autoRecon	yes	
detect!	detected	
prio!	prioTarget	
announcer	yes	
applyMarks	yes	
mrgs	no	

So, when the mission starts up, with above settings, all blue aircraft are added to the recon roster when they appear.

So now we want to selectively remove two aircraft from active scout duty: the AI Falcon and the ‘Blind Frog’, both located at Kutaisi.



We simply placed a Trigger Zone (‘I AM BLIND!) with the ‘scout’ attribute over these aircraft, and set the value to false. When these planes enter the game, Recon mode ignores them. Even if you enter the Blind Frog’s cockpit multiple times later in the game, it will never be granted scouting abilities. That is why neither the F-16 (also in the ‘blind’ trigger zone) nor the Blind Frog ever report the enemy ground forces east of Kutaisi.

Also note that there is a trigger zone named ‘Player Eyes’ at Senaki Kolkhi, encompassing the two blue player aircraft and that explicitly enables the two player aircraft there as scout. With the current config settings, that is redundant, as blue aircraft are always added as scouts. However, should you later change the config to exclude blue aircraft from the scout roster, those two will still remain scouts.

8.23.3 Discussion

So here are some fun things for you to try with this mission. Let’s see how you do the following:

- Change the reports to give location info in MGRS
- Change the “Primary contact discovered…” message to include Lat and Lon in addition to Time
- Turn on the silent flag for the “Prio Targets” zone and try to predict what will happen. Did you expect to still receive the ‘Gotcha’ message? Why did it still appear?
- Change the mission so the Recon Mode switches Off when the “Oh snap, it’s a trap’ message is received.
- Remove all player planes from the scout roster

8.24 Owned Zones ME Integration.miz

8.24.1 Legacy Warning

This demo uses the pre-2.0 “legacy” version of owned zones which remains available in DML. The legacy version essentially combines the functionality of owned zones (capturable zones) and unit production (defenders and attackers). This demo’s objectives remain unchanged, the (very brief) sister tutorial “Owned Zones and Factories’ highlights the changes that the new split owned zones/factory functionality engender.

You can still use the “legacy” version of ownedZones (remains still part of DML) to fully re-create this demo and use the original functionality of the module as shown here.

8.24.2 Demonstration Goals

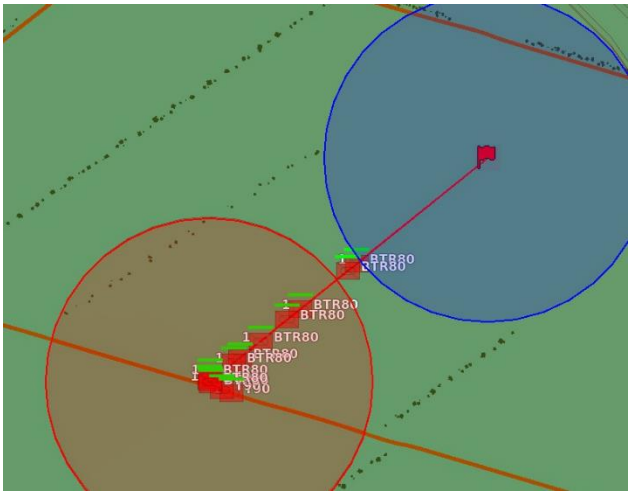
This mission shows how Owned Zones work in general and how they can be used to set ME Flags. It also offers a nice test bed to illustrate how the various cooldowns work and can influence the game.

8.24.3 What To Explore

8.24.3.1 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

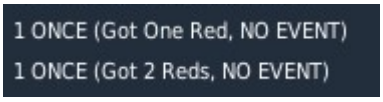
8.24.3.2 ME

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

Name	Value
owner	blue
defendersBLUE	Soldier M4
spawnRadius	5
attackersRED	BTR-80, BTR-80
defendersRED	Soldier M4
attackRadius	10

the owned zones defenderBLUE attribute: “Soldier M4”. Defenders are spawned at mission start.

After capture by red we produce the same unit, this time as defendersRED. Note that owned zones can spawn usually ‘blue’ units as red. Later, the BTR-80s are the units that we defined under attackersRED. Note that there are no attackersBLUE defined: this zone does not produce attacking units.



Note the two triggers “ONCE Got One Red” and “ONCE Got 2 Reds”). They both trigger on Flag 10, one when Flag 10 has a value of 1, the other when the value of Flag 10 is 2. If you inspect the ownedZonesConfig, you will find that we are banging Flag 10 for red: f!=10. Each 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.

8.24.4 Discussion

Again, this mission requires 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.

8.25 FARP and away.miz (tbc)

8.25.1 Demonstration Goals

8.25.2 What To Explore

8.25.2.1 In Mission

8.25.2.2 ME

8.25.3 Discussion

8.26 Keeping The Score: Player Score.miz

8.26.1 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.

Note:
This demo shows only the score reporting abilities of the PlayerScore module. The 'feats' reporting ability is turned off. See "More score" and "Later Score" for PlayerScore's expanded reporting

8.26.2 What To Explore

8.26.2.1 In Mission

Start the mission and use the (free) Su-25T or one of the A-10 (A or C) to lay waste to the poor targets on the ground. If you fly the C-Hog, there's also target lasing available with a code of 1688.

When you kill a ground unit on the main runway, note how your score increases each time the unit is killed (if the units is still "cooking off", the score is awarded only after the unit explodes). Hitting a BTR-80 yields 10 points, while a Leo nets 30.



Close to the main tarmac, marked by red smoke, are three T-90 tanks. Kill them all, and watch the score. After killing one of them you get a message that you killed a strategic target ("Big Kahuna") and receive a significantly higher score (150). Note that scores are totaled as well.

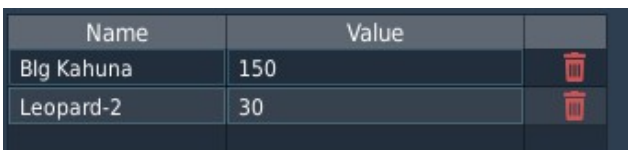
After successfully killing some vehicles (at least one BTR-80 or Leopard, and one T-90), choose Communication→Other… →Score/Kills. This is the Player Score UI module that allows a player access to more detailed personal score information. You are presented with your personal kill log:

- Total score and number of kills
- List of all types killed and their number
- Note that named kills also appear as a separate type



8.26.2.2 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

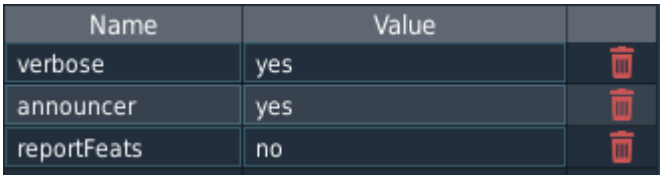


unit score", as only units that match that name receive this score, and since unit names must be unique in DCS, there can only ever be one unit that is awarded that score.

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.

8.26.3 Discussion



PlayerScore can also report feats that players accomplish. Feats are either placed by mission designers, or are reported by other DML modules (e.g., HeloTroops or CSARmanager). By default, PlayerScore always reports all accomplished feats as well as scores. Since this mission does not feats, reporting feats would needlessly clutter the display, and I turned that off in the config zone. Feats are explored in the demos "More Score" and "Later Score". Please see those chapters.

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.
- Turn on reporting of feats to see how they are reported

8.27 The Zonal Countdown (Local/Global Flag Demo)

8.27.1 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!

8.27.2 What To Explore

8.27.2.1 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.



8.27.2.2 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

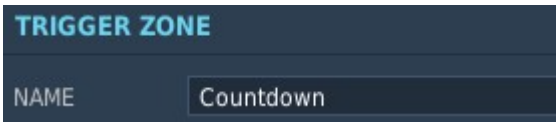
Once the pulse starts beating, it feeds into a **count down** module. This module starts counting at 11 (‘this one goes to eleven!’), and every time a signal is received on local *pulse, it counts down on. The current value of the counter is passed out on local *cVal for anyone who is interested (the 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. Then it assembles the message to put on the screen from message “*This is a text from Zone <z>. Countdown: <v: *cVal>”* and resolves any wildcards it finds. The two wildcards *<z>* and *<v: *cVal>* are replaced with the zone’s name (“Countdown”) and value of the (local) flag *cVal (which is the counter from the count-down above).

messenger?	*counted
message	This is a text from Zone <z>
clearScreen	yes

Before it's put on the screen, the screen is cleared, removing all previously displayed messages, and moving the output line to the top.



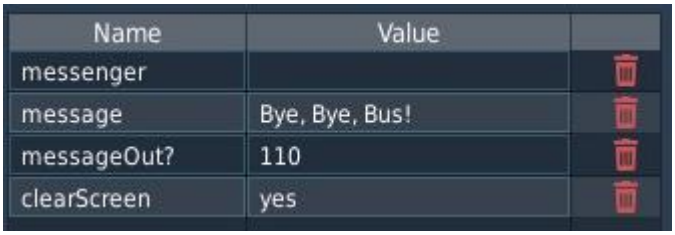


This is what makes the countdown seem to be updated; in reality, it's merely overwritten.

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



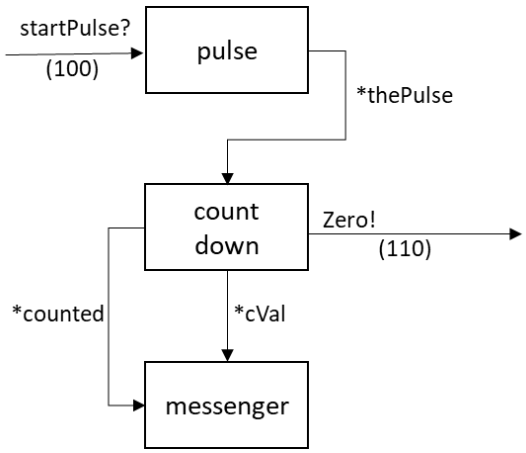
At the same time as ME is busy blowing up the bus, the messenger in the "Zero Message" zone gets active: triggered by the same signal on (ME compatible global number) flag 110, it clears the screen (erasing the last count down message), and writes Bye, Bye, Bus!" to the screen. This is a purely cosmetic 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 😊.



8.27.3 Discussion

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.

8.28 Frog Men Training.miz

8.28.1 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



8.28.2 What To Explore

8.28.2.1 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.



8.28.2.2 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.

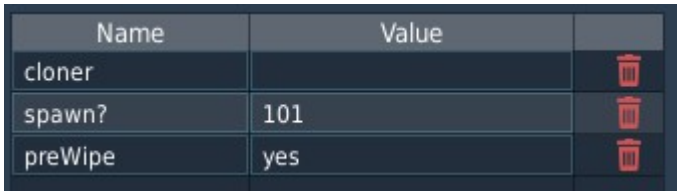
ENDLESS SPAWNS
This is achieved by creating a feedback loop from *empty!* into the *in?/spawn?* input of the cloner. Whenever the last unit of the previous clone is destroyed, *empty!* fires, which feeds into *in?/spawn* creating a new clone cycle.

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

Name	Value	
pulse!	101	
paused	yes	
activate?	100	
time	300	

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!* → *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).

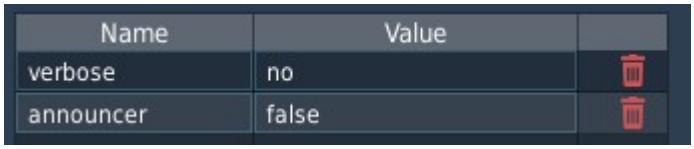
One possible (there are many others) solution is to pause the pulser (*paused yes*). This prevents the pulser from starting any pulses. When the SAM Hunters appear, ME sets flag 100 to one, which is detected by the pulser's *activate? 100* input. This activates the pulser, which causes it to immediately 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.

8.28.3 Discussion

To make this mission more interesting, change Guardian Angel's configuration to not destroy the missiles, and disable respawning for the SAMs.

8.29 CSAR of Georgia.miz

8.29.1 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.

By default, CSAR manager only allows certain helicopters to be used for evacuation (the 'troop carriers', which while I write this are the Huey, Hip and Hind). This demo also changes this by allowing the Gazelle Minigun configuration and Huey, and disallowing all others.

8.29.2 What To Explore

8.29.2.1 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:

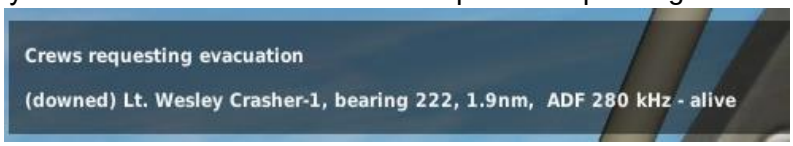
- *List active CSAR*
  This shows a list of all currently available (waiting for pick-up) CSAR missions, along

with their ELT's frequency. If vectoring is enabled (by default), this also shows range and bearing to each of the evacuees.

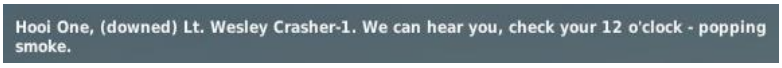
- *Status of rescued crew aboard*
  Shows the medical status of each evacuee that you have picked up. This is meant for later extension when time-critical missions are supported.

- *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.

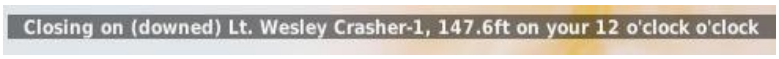
Choose "List active CSAR", and you will receive the news that one pilot is requesting extraction: a Lt. Wesley Crasher. They are some 1.9 miles away at bearing 222, and their ELT can be located at a certain frequency (here 280 KHz). This frequency can change, as this particular CSAR is set up with randomized 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 (which triggers 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 (instead of one). 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.

8.29.2.2 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.

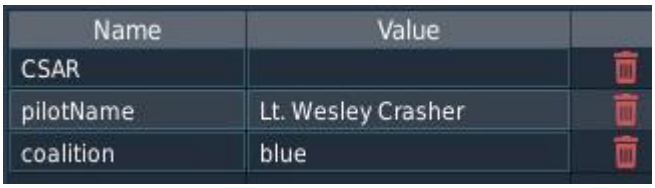
For CSAR Manager to work, it requires two important items: the Base (or bases, as CSAR Manager happily accepts multiple bases) where you return downed pilots / evacuees to, and the evacuees themselves.



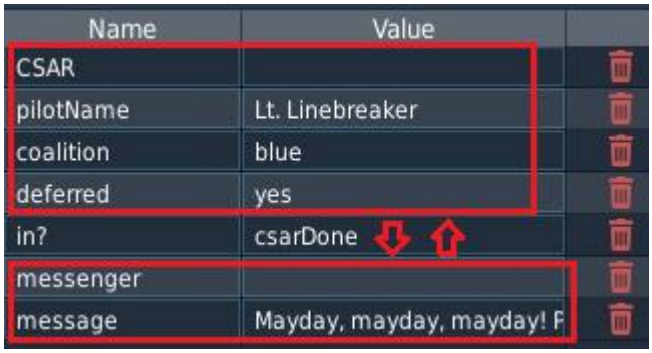
Turn your attention to Senaki-Kolkhi. And note the light blue quad-based trigger zone “CSAR safe”. It only has a single attribute: CSARSAFE, which marks the entire contents of that trigger zone as an area that, if you touch down with your helicopter inside it, and have evacuees loaded, their CSAR missions complete successfully. Whenever you include the CSAR Manager in a mission, remember to place at least one CSARBASE 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.



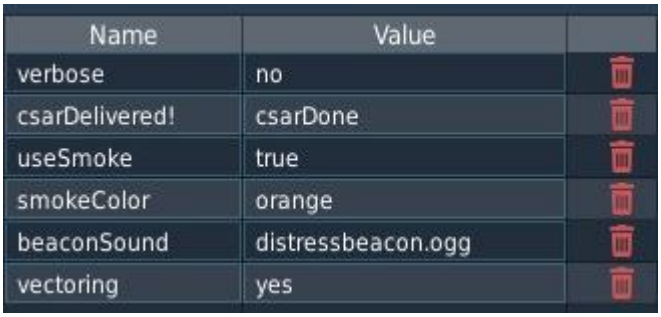
- With a deferred CSAR Zone. These are like the CSAR zone above with an additional important distinction: they do not create their CSAR Mission when the main mission begins, but wait for the flag startCSAR (or one of its synonyms) to change, and then create a CSAR Mission. Since they are controlled by 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:

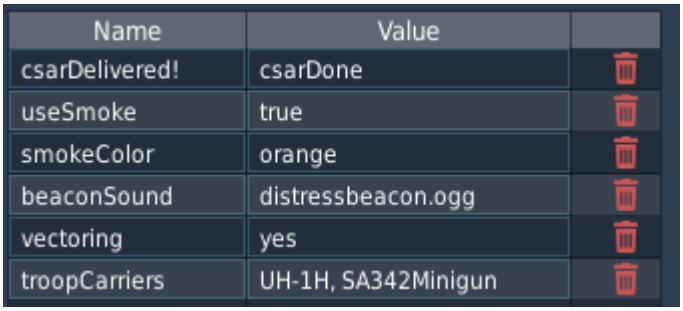


CSAR Manager supports a number of convenient flag banging abilities when a player returns a pilot to a CSARBASE: one for red ("csarRedDelivered!"), one for blue ("csarBlueDelivered!"), and one for any successful CSAR ("csarDelivered!"). We use the latter, and wire it into Accident Site-3's "in?" flag. So every time a player safely delivers a pilot to base, a new CSAR mission is created at Accident Site 3.

8.29.3 Discussion

Since this module's config zone offers so many cool features, let's experiment a little. Change the existing config accordingly:

- 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
- Restrict all CSAR missions to Huey and Gazelle Minigun



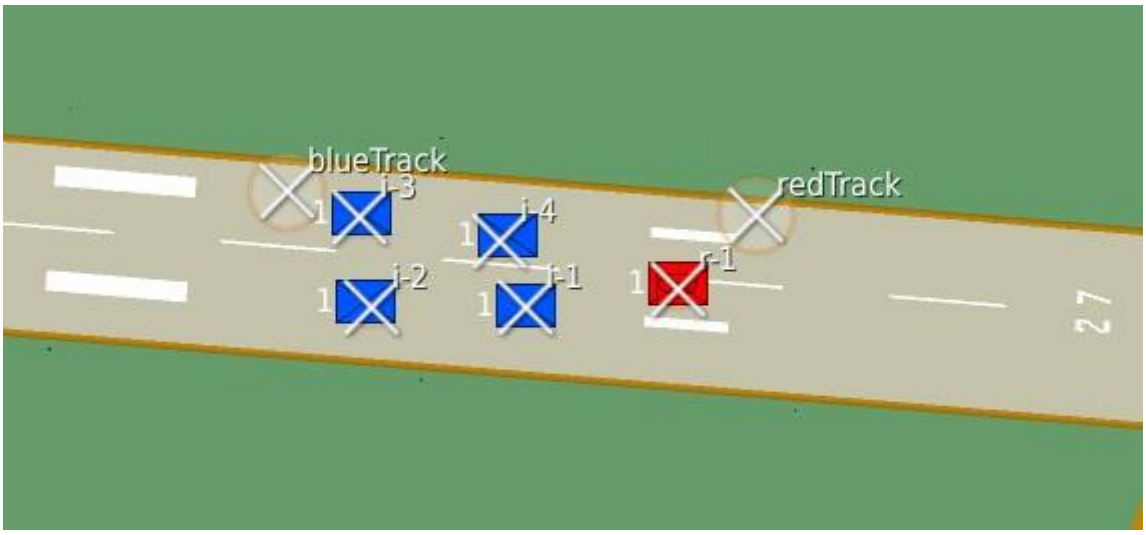
And now for a real challenge: Have your mission create new random CSAR that whenever you successfully bring back one pilot. To make this more manageable, let's add the following conditions:

- 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.

8.30 Track This! (Group Tracker)

8.30.1 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.

8.30.2 What To Explore

8.30.2.1 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.

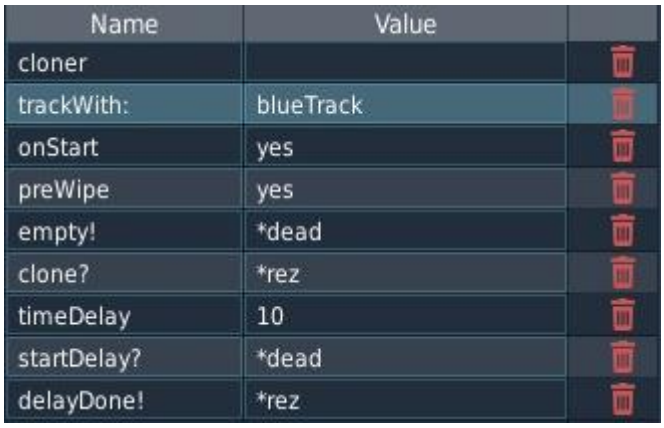
Somewhat more intriguing are the scrolling text messages on the side that accompany the grisly spectacle: a kill count that is updated each time one side makes a kill.



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.

8.30.2.2 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:



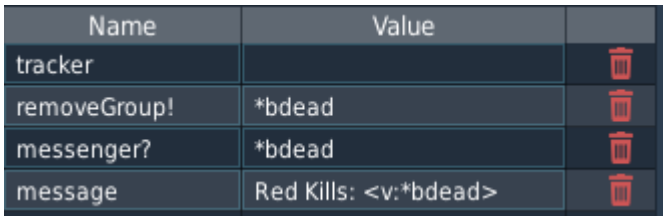
Name	Value
cloner	
trackWith:	blueTrack
onStart	yes
preWipe	yes
empty!	*dead
clone?	*rez
timeDelay	10
startDelay?	*dead
delayDone!	*rez

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 self-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?	*bdead
message	Red Kills: <v:*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 updates 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 number of increases is done on the removeGroup! flag). This output is wired to the (local) *bdead flag.

So groupTracker 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 messenger? 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: <v:*bdead>". When it encounters the "<v:*bdead> wildcard, Messenger then looks at the value of local *bdead flag. Messenger now loads the current value of that flag, and exchanges that value for the 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.

8.30.3 Discussion

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

8.31 Watchflags Demo

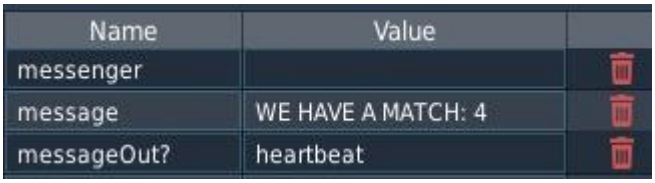
8.31.1 Demonstration Goals

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 patterns emerge: often, being able to narrow down a trigger's change event can 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.

8.31.2 What To Explore

8.31.2.1 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:

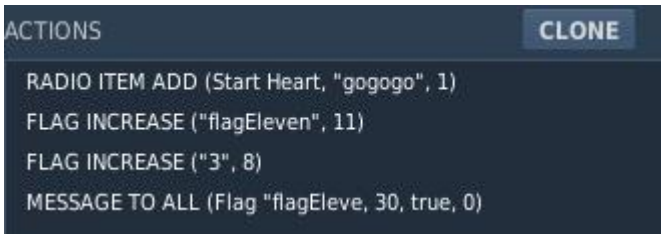


That is the last message we are looking for, and after you have confirmed that the message appears, you can close the mission.

8.31.2.2 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)

Name	Value
pulse!	heartbeat
startPulse?	gogogo
time	5
pulseMethod	inc
pulseStopped	yes
messenger	
message	Pulse Started
messageOut?	gogogo

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').

Name	Value
messenger	
message	Change detected
messageOut?	heartbeat

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.

Name	Value
messenger	
message	Inc -- Count = *value
messageValue?	heartbeat
messageOut?	heartbeat
triggerMethod	inc

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

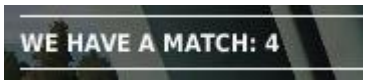
When you check the game’s output more than 20 seconds after you start the heart, you’ll notice that the line “<5 – Count = …” no longer appears.



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	Value
messenger	
message	matches FlagEleven!!!
messageOut?	heartbeat
triggerMethod	=flagEleven

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.



8.31.3 Discussion

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. 😊

8.32 Viper with a double youu (Wiper)

8.32.1 Demonstration Goals

Wiper is a module that removes a variety of objects from the game. This demo runs through its most important uses, and shows some of its limitations.

8.32.2 What To Explore

8.32.2.1 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, fire and smoke remain (but the LUV’s debris is cleared out).

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

8.32.2.2 ME

This mission is more about the limitations of wiper than its abilities. Wiper is universally and easily understood: if 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.

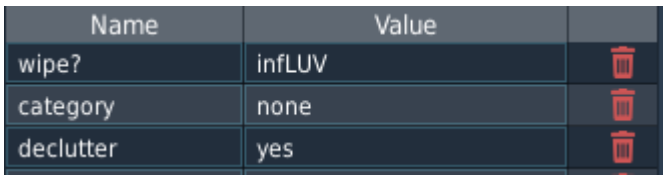
THE HAPPY CASE

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 wipeNamed attribute to selectively remove objects from a larger group of possible wipe targets.

WIPING DEBRIS / 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.



A design quirk in DCS differentiates not only between units, objects and scenery, but has a another, completely separate internal category for debris like wrecks, craters, and fire that result from destruction. Wiper can remove some of them, and it requires the separate attribute "*declutter*" to trigger that option. This is intentional, since currently, the API that DCS provides for this function is experimental and might be linked to some server crashes.

So, while DCS can and will remove debris like wrecks, it will currently not remove smoke nor flames. This is shown in the unfortunately named “Tank Respawn” demo (the LUV respawns, not the tank). Note that all debris are removed, but the flames remain. Note also the ‘category = none’ attribute that turns off object wipes for slightly better performance.

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 packaged 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.



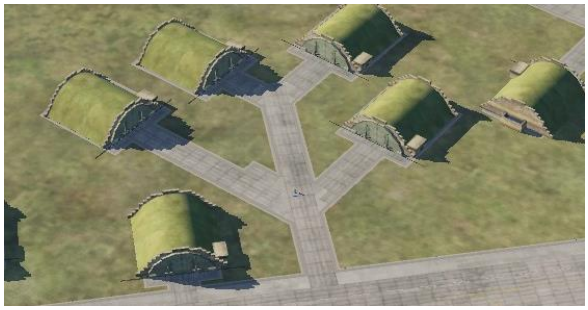
8.32.3 Discussion

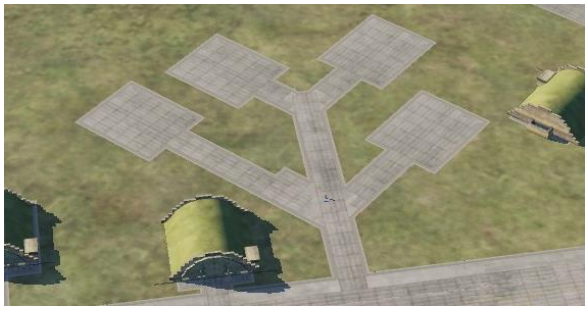
Wiper has another ability that comes in handy when dealing with scenery or heavily populated areas.



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.



8.33 Radio Go Go (Radio Trigger)

8.33.1 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.

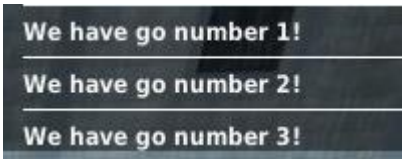
To remedy this, the Radio Trigger module provides a way that notices a flag change in the input, generates a signal on the out! flag, and then resets the input flag so that when the player chooses it again, a new signal is generated.

8.33.2 What To Explore

8.33.2.1 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.

8.33.2.2 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.

8.33.3 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.

8.34 xFlags – Field Day (Decisions, Flag Testing)

8.34.1 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.



8.34.2 What To Explore

8.34.2.1 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, messages appear. Take note of what they say, and once all race cars have stopped their engines, you can exit the mission

8.34.2.2 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 through 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 is in contrast to 'One of us' that merely requires a minimum, while exactly requires an exact match.

Name	Value	
xFlags?	oneA, twoA, threeA	
require	exactly	
#hits	2	
xSuccess!	*hit	

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.

Name	Value
xFlags?	oneA, twoA, threeA
require	more than
#hits	1
xSuccess!	*hit

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'.

Name	Value
xFlags?	oneA, twoA, threeA
require	at least
#hits	1
xSuccess!	*hit

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.

Name	Value
xFlags?	oneA, twoA, threeA
require	most
#hits	1
xSuccess!	*hit

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.

Name	Value
xFlags?	oneA, twoA, threeA
require	half or more
#hits	1
xSuccess!	*hit

8.34.3 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

Name	Value
xFlags?	oneA, twoA, threeA
require	any
xReset?	*hit
xSuccess!	*hit

line (or rather, enters it's unitZone, which then triggers its arrival flag.

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.



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.

8.35 Virgin (Civ) Air / Air Caucasus II / One-Way Air (CivAir)

8.35.1 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 it can also spice up a mission by providing aircraft that must *not* be shot down.



8.35.2 What To Explore

There are two missions here to explore: Virgin (Civ) Air, and Air Caucasus II. We'll discuss both.

8.35.2.1 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, wait for 5 seconds (civAir allows the mission to spin up for 5 seconds before it starts its own magic), and then 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 view and zoom out. The white icons all depict a civilian flights under civAir's control. Note that the aircraft types are fully randomized, and the flights fly all over the Caucasus map. Also take note of the diversity of liveries.

Now switch to mission 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. They never fly to any other destination or depart from another airfield.



Also note that there are fewer different aircraft types compared to Virgin (Civ) Air: there are only the

- Yak-40,
- C-130 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)

Note also that the Yak-40 is by far the most common aircraft type that is used in flights.

8.35.2.2 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 lacks some atmospheric sugar (CivAir itself uses negligible processing power, the AI planes only very little).

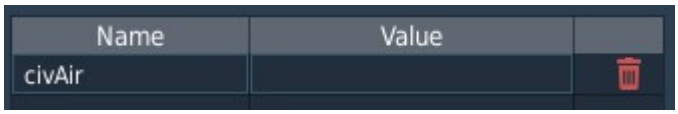


Air Caucasus II demonstrates how to customize civAir in a few seconds by adding to the map:

- We add a trigger zone close to the airports of Batumi, Kobuleti, Kutaisi, Senaki-Kolkhi and Sukhumi-Babushara
- To each trigger zone we add just a single attribute: civAir.

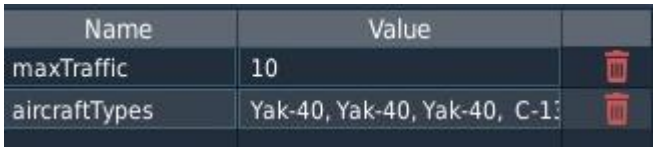
When your mission now starts up, civAir detects these zones, and from those it builds a list of airfields to use-

In the Virgin (Civ) Air demo, no such zones were present, and Civ Air then proceeded to use all the airfields on the map.



Note that the civAir attribute has no value, telling the module that the closest airfield to this trigger zone is a viable departure and destination. You can change that selectively by using special keywords:

- `closed` will disallow this airfield for both arrivals and departures
- `depart`, `start`, or take off will flights only to depart from this airfield
- `land` or `arrive` makes this a landing only airfield (no departures) .



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, we have noticed that the Yak-40 small business jet is much more prevalent than the other types. This is because in the aircraftTypes attribute, that type Yak-40 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).

8.35.3 Discussion

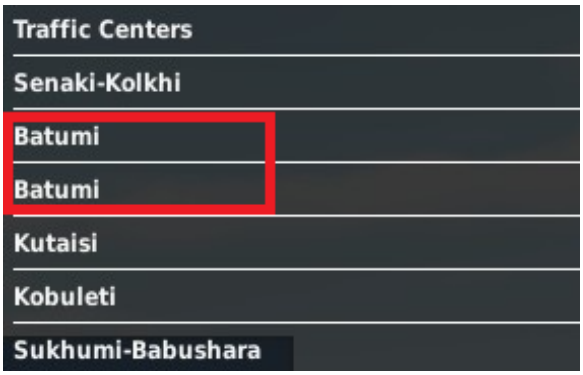
As you can see, civAir works well straight out of the box, and can easily be customized to use only a few airfields and airframes simply by adding some zones and attributes

8.35.3.1 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.

8.35.3.2 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.

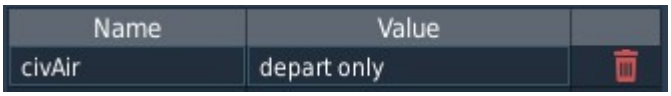
8.35.3.3 Increasing Traffic

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.

8.35.3.4 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.

8.36 Count Bases Blue (baseCaptured, xFlags)

8.36.1 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.

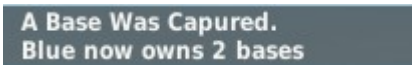


8.36.2 What To Explore

8.36.2.1 In Mission

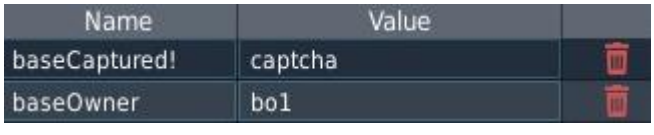
Enter the mission, and watch how the various Leo tanks approach their target FARPS. After a while, three blue FARPs (from a total of four) are being captured.

Each time a FARP is captured, a message appears with a new total of FARPs that blue still holds.



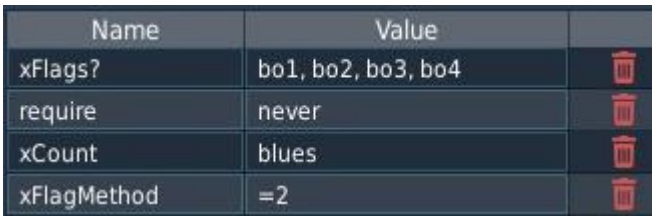
8.36.2.2 ME

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).



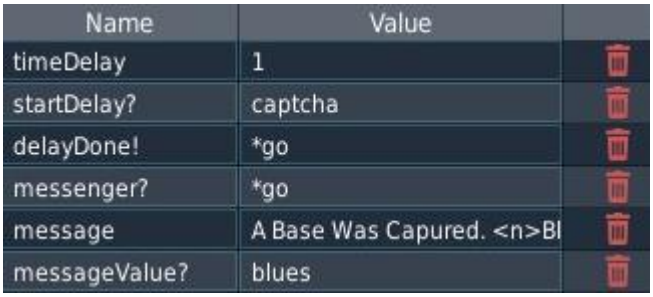
Now, these boX flags are all collected by an xFlags module, configured to function as a counter:

- 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)

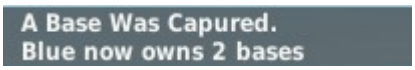


- Require is set to 'never', meaning that the xFlags module will continuously run, and never stop, no matter what configuration bo1-bo4 have. This is important since 'require' defaults to "some", so if we did not change it to 'never', xFlags would stop after the first base is owned by blue.
- xFlagMethod 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.



8.36.3 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

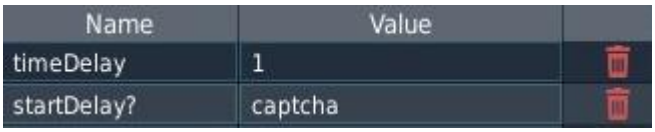
So why did we set 'require' to 'never'? Remember that xFlags pauses once the requirement is met, and dafaults to 'some'. In default configuration, the first time that a base is owned by blue – which is the condition the mission begins at. Since always want xFlags to supply us with the number of bases 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?

8.37 Pilots at their Limits (Limited Airframes)

8.37.1 Demonstration Goals

This mission demonstrates how Limited Airframes works when players switch or crash planes. We'll walk through the following scenarios:

- Changing airframes in a safe zone
- Changing airframes on the ground outside a safe zone (ditching on the ground)
- Crashing the plane
- Ejecting from a plane
- Ditching an airframe in mid-air



8.37.2 What To Explore

8.37.2.1 In Mission

At the start of the mission, enter Frogger One SAFE, sitting cold on the ground in Senaki-Kolkhi, a pilot-safe zone. Now, change into Frogger Two SAFE's slot. Note that since you are leaving a plane in a pilot safe zone, you can change airframes without an issue.

Now change into the slot UNSAFE Frogger, which is sitting in an unprotected area. Note that again, you can change *into* that plane without an issue because the plane you are leaving is sitting safe on the ground, in a pilot-safe area.

Now change back into Frogger One SAFE.

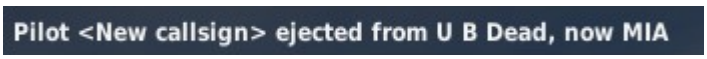


Since you are changing into a different air frame outside of a pilot-safe zone, you'll lose one pilot (if CSAR Manager was active, it would have generated a CSAR mission for where that plane was standing).

Now change into "U B Dead", an SU-25T flying above mountains with an



empty fuel tank. Crash the plane into the ground. Limited Airframes announces your death and deducts another pilot from your pool.

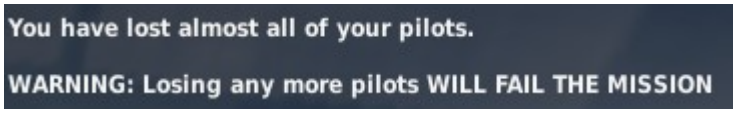


Re-enter U B Dead. Try to cheat death by ejecting. Again, one pilot will be deducted. If you had CSAR Manager installed, it would have generated a CSAR Mission for the pilot.

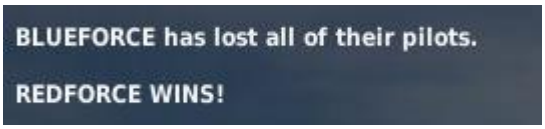


So let's try and be creative. Re-enter U B Dead, and quickly switch to another plane, for example Frogger Safe One, before it flies into the ground. Nice try, but Limited Airframes still deducts a pilot, and if CSAR Manager was installed, another CSAR Mission would be created.

Enter one of the SAFE Froggers, and eject. Note that even though they are in a safe zone, your side loses a pilot. Again, if CSAR Manager was installed, a CSAR mission would be started.



Go on, and waste more air frames. Once you reach your last pilot, you get a warning

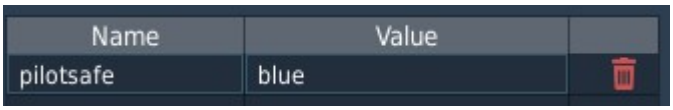


Once you lose that, it's curtains for the mission when you re-seat (in multiplayer, you can opt not to re-seat, and have another player try and rescue one or more pilots – when CSAR Manager is active – or simply wait for the mission to conclude).

8.37.2.2 ME

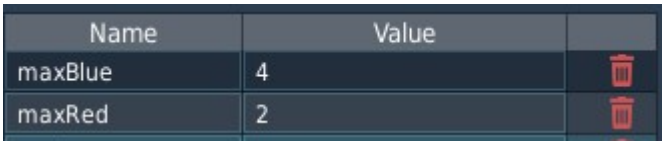
In ME there are only a couple of interesting things to observe:

- Senaki-Kolkhi has a Quad Trigger Zone "Senaki Blue Safe" with the *pilotsafe* attribute, and 'blue' as its value. This means that *only blue planes* are safe to switch here because it only mentions blue (if none were mentioned, all planes would be safe here)





- This mission also shows how to set up Limited Airframes asymmetrically for red and blue



with a configuration zone. Note how there are different values for maxBlue and maxRed, giving blue an advantage of two more pilots. You can use this to balance a game where one side has better (more capable) aircraft than the other.

8.37.3 Discussion

CHALLENGE
So let's see what happens if you take one of the **RED** UNSAFE Froggers, fly it to Senaki-Kolkhi, and park it inside the pilot-safe zone, then switch planes. What did you expect would happen?



8.38 Gate and Switch (Changer – Pulse & Gated Switch)

8.38.1 Demonstration Goals

A common design pattern in missions is where you need to trigger one course of action one condition is true, and other if it isn't.

Gate and Switch demonstrates a simple way how to do this with the Changer module, and with the DML 'pulse' method.



In this mission we want to be able to command spawning F117s that take off and fly away – but only when our plane is in a certain zone. When we are outside the zone, commands to spawn planes will only result in a message.

8.38.2 What To Explore

8.38.2.1 In Mission

Enter the Frogger and look outside. In front of you there is a big circle marked out with tires and flags. Before we move the aircraft, go to Communication → Other → Spawn And Depart.

A message comes up to tell you to move your plane into the circle. Try this a few more times. The result is always the same.



Ok, let's comply. Move the Su-25T into the circle marked out by tires and flags, come to a complete stop, and try the communications menu again. This time, an F-117 spawns on the runway, and after a brief time, it takes off, heading for another airfield. Wait until it has taken off and try the command again. A new F-117 spawns and takes off.



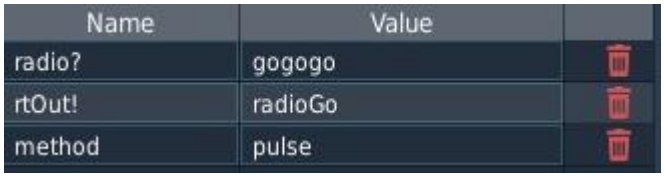
Taxi your aircraft outside of the circle. If you have given multiple spawn commands inside the circle, wait until all scheduled spawns have completed and no more planes spawn.

Then try the command again. You should again see the message requesting you to move into the circle.

8.38.2.2 ME

Now let's see how this is done in ME. There are multiple challenges:

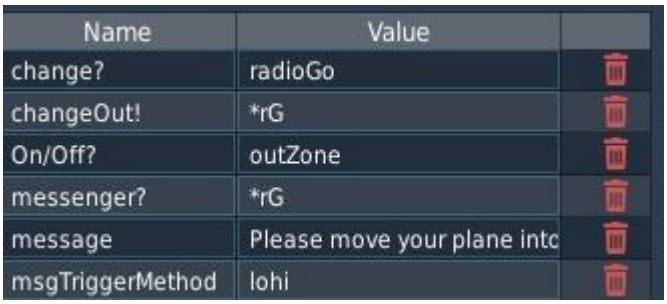
- Create a way to connect two different actions to a repeatable communications menu item
- Have only one of those actions activate depending on a game condition (here: player's plane inside a zone / outside of that zone)
- Ensure that the radio commands do not 'bleed' across actions when the condition changes



The first one is simple to implement, since we have a module especially built for this: Radio Trigger. Note, however the peculiar method we chose: 'pulse'. This method sets the output flag rtOut! to 1 for 3 seconds, and then sets it back to zero. We'll discuss later why. Let's just remember that an output set up this way should connect to an input configured for 'lohi' activation.

The rtOut! signal feeds via the radioGo flag into two zones "Respond to radio Outside" and "Respond to radio Inside" that are configured almost identically:

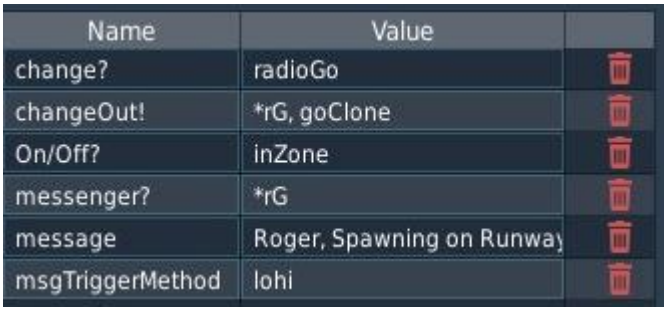
- A changer module reads the radioGo signal and passes it unchanged to local *rG flag.
- The local *rG flag activates the messenger flag to output a message.



We have modified this basic functionality in multiple important ways:

SWITCH

Changer uses the On/Off? Input to control its gate. This means that whenever the value on the flag that connects to On/Off? is zero, the gate is closed, and no signal propagates from radioGo to local *rG. When, on the other hand, the value on outZone is anything other than zero, the gate is open, and signals can flow through the changer to the messenger. Using a changer as a "gated switch" (to block or allow information to flow) is an important and common design pattern that you can use in your missions to great advantage.

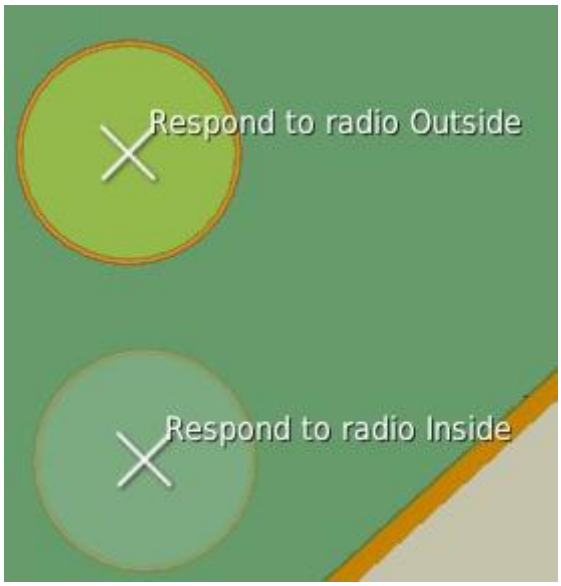


If you inspect the second zone that is configured almost identically, you'll find that their only difference is that their gate is controlled by different flags: outZone and inZone. This means that these two modules can be active at different times (see later)

Note also that this zones gate switch (changeout!) provides signals on two different flags: local *rG and global goClone. We'll use that signal later.

PULSE AND LOHI
When the information can flow through the gate above, it arrives at the messenger. Remember that the signal originates from the radio trigger that converts giving a radio command into a pulse, making the flag's value mostly 0 (zero), and after an activation 1 for a few seconds. The 'lohi' method to trigger is very useful as it only triggers on a flag change that goes up, but never when the flag resets back to 0. Additionally, it only activates specifically when the flag value rises from 0 – so 'bounces' (erroneous additional increases on the flag, i.e. multiple simultaneous activations) are filtered out.

HOW TO NOT CROSS-BLEED INFORMATION
But why go to all this trouble with 'pulse' and 'lohi'? The answer to this lies in the fact that multiple modules are listening to the radio trigger. Let's try and walk through a normal setup where the radio trigger is set up to simply increment the output flag 'radioGo'. The gate for zone "Outside" is open, the gate for "Inside" is closed.



Now the player chooses the radio item, and radioGo is incremented from 0 to 1. Since "Outside" gate is open, the signal propagates to the messenger and the message "Please move…" is displayed. Since "Inside" gate is closed, the radioGo signal does not propagate, and that messenger remains silent. The player repeats this a couple of time. Each time, the Outside messenger fires, the Inside messenger stays silent, and at the end, the value of radioGo has increased (through four 'inc' method invocations= to 4.

Now the player moves into the circle. Among other things, this activates Inside's gate. Now radioGo's value (which is 4) flows through the changer to messenger. Messenger compares this value with the last (zero, it has never received a signal before), and immediately (and erroneously) triggers. This is information crossbleed – part of the information of previous activations have made it through the gate. This we want to avoid.

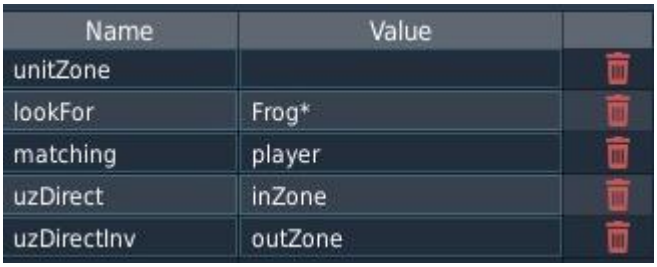
The solution is to use a flag that only carries information transiently – for a brief time – and then returns into its default inactive state (zero). That's what the pulse method does: it briefly (for a few seconds) goes to 1, and then returns to zero. **Therefore, 'pulse'/'lohi' is the ideal method combination for gated flags.**

That is why the radio trigger issues pulses, and the messengers are cued for 'lohi' signal transition: shortly after the signal is received, it falls back to zero, the information cannot easily cross-bleed to other modules.

Note:
I intentionally wrote 'easily' above – an edge case remains: we can imagine a situation where the player enters the circle and enables Inner's gate in the brief period where the radioGo flag is still 1. This can happen, but very unlikely. You can try to guard against this

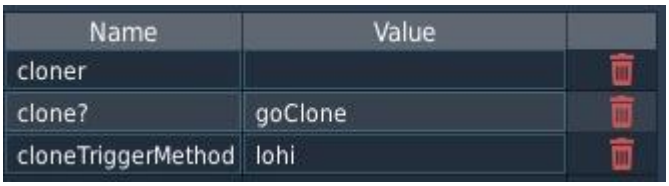
using delayFlag modules, but usually guarding against these cases is not worth the effort unless the result is game-breaking.

DIFFERENT ACTIVATIONS



So how do we achieve that the “Inside” and “Outside” gates open and close at the right time? We use the unitZone’s uzDirect and uzDirectInv outputs that carry the exact signals that we are looking for: inZone is 1 if, and only if, the player is inside the zone, and 0 otherwise. We use this to control ‘Inside’ gate. uzDirectInv is 1 if, and only if, the player’s plane is outside the zone, and 0 otherwise. We use this to control the gate for Outside. Since they switch values at exactly the same time, there is no danger of overlap where both gates are enabled or both gates are disabled.

THE CLONER: SIMPLE SWITCHED GATE



The Inside switch (changer) has two outputs, of which the global goClone flag is used to trigger the cloner. That module’s setup matches the exact use pattern for gated modules: a gated “pulse” input signal

8.38.3 Discussion

CHALLENGE

What would be the result if we removed the clone zone’s “cloneTriggerMethod” attribute? Why does this happen?

CHALLENGE

We could have used RadioTrigger’s radio? inputflag “gogogo” as input for the changers and lived without rtOut! and the pulseMethod as long as radio? is connected. Can you find out why this is true?

CHALLENGE

Above, I wrote that an output set to ‘pulse’ method should preferrable connect to an input configured for ‘lohi’. Technically, ‘inc’ will also work with an output configured for ‘pulse’. Can you find out why, and under which conditions this is true?

CHALLENGE

It’s happy coincidence (and if you believe that…) that unitZone happens to have both uzDirect and uzDirectInv available. But what if you used a module that only holds one output

and you needed both the normal and inverse signal? How would you build two stacks that should flip like our example above without overlap? Hint: you need multiple instances of a module that you are already using, but in a different configuration.

8.39 Good Grief (unGrief)

8.39.1 Demonstration Goals

The internet is full of great people. Unfortunately, the internet is also chock full of asocial elements that love to spoil other people's fun – griefers. If you host missions on your server and see these trolls showing up, one possible recourse is DML's unGrief that may act as a low-impact, "low-grade griefer" deterrent (determined griefers will keep showing up and must be banned by IP).

This demo shows how unGrief reacts to a player who is engaging in the griefing activity of intentionally killing their own side.

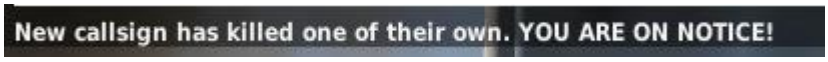


8.39.2 What To Explore

8.39.2.1 In Mission

In this mission there are various blue frames and a red airframe. Start by boarding a blue airframe and kill a blue ground unit. If you own the UH-1 module, this can be easily accomplished by using the gunner positions, else use the Frogger, or (if you own FC3) the Hog.

When you kill the first blue unit, a warning appears. This is regarded as an accident, and your initial kill is forgiven, but you are on notice:



Kill a red unit. No warnings appear. This is a legal enemy kill.

Now kill another blue unit. This time, unGrief assumes that you are acting in bad faith, punishes you, and gives you a final chance:



Your aircraft blows up and/or crashes.

Pick a new blue plane.

Kill another blue unit. No assumptions of accident. You plane explodes.

But there's more: Your third friendly kill closes the deal for unGrief: you are a griefer. Since it's set to 'wrathful', it will not forgive you. From now on, you are dead, and unGrief will make sure you stay dead. You have overstayed your welcome:

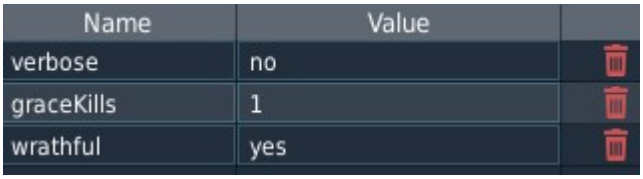


Whenever you try to slot, it adds 100t to your plane's internal cargo, and ignites 1kg of explosives inside your plane. **Even if you try to change sides**. Try to enter the red Frogger to experience this. If you want to be able to fly this mission again, it must be restarted first.

8.39.2.2 ME
All of unGrief's features are controlled from within a single config zone.



Usually, all you need to do is add unGrief (and its dependencies) to your mission, and you are all set. With a config zone you can set details like grace kills and enable your wrath.



8.39.3 Discussion
unGrief is near hassle-free and easy to set up, yet there are still some things to try out:

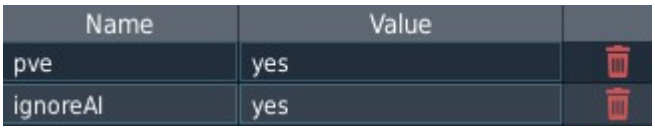
Zero graceKills
What will happen when you set graceKills to zero? Pretty much what you expect. From experience, setting graceKills is only a good choice for 'strict' servers. If your server gravitates to more relaxed, casual play-styles, you may want to allow at least one strike before retaliating.

The Grapes of Wrath
This demo has unGrief's "wrathful" enabled, in effect removing a repeat offender from this mission until it is restarted. Disabling is easy: simply set it to false, or remove the attribute entirely from the config zone. Be mindful using wrathful, as it may make a mission unplayable with casual players.

The Fluster Cluck
unGrief responds to 'killed' events. When a unit is killed, it inspects who is responsible for the kill, and when they are on the same side, investigates. This means that a cluster bomb dropped too close to your own units – especially infantry – can immediately send you over the grace and wrath limits, so be careful when unGrief is watching you!

Also, some crew AI (e.g. George and Petr) do not differentiate between friend or foe when offering up targets, so be very sure whom you engage, lest unGrief retaliates.

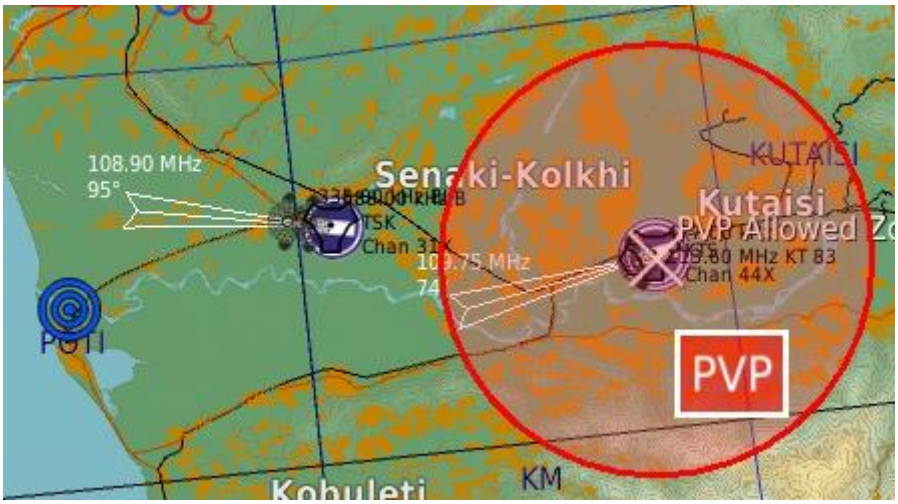
PVE rules (no player-player killing)
A common griefer tactic is player-killing. To combat that, you may simply want to set your server up to disallow player-player kill, but allow everything else. To do so, simply set the pve attribute to true, and add the ignoreAI attribute with a value of true.



8.40 The Danger Zone (MP – PvP Zones in PvE)

8.40.1 Demonstration Goals

The unGrief module allows dedicated PVP zones in a PvE environment. This demo shows how easy it is to add PvP zones to a multi-player mission that is set up as PvE.



8.40.2 What To Explore

Note: This is a multiplayer mission, it's quite pointless to fly it solo as you can't kill anyone.

8.40.2.1 In Mission

One player should enter the red, the other the blue Enter the blue PVE Frogfoot. Both continue straight on. Blue should shoot down the red Frogfoot. Blue receives a warning that the next instance of PVP kill will have consequences.

Now both players should change into the respective PVP planes. These planes are both inside the PvP zone, where PVP killing is allowed. As soon as you enter the cockpit, you are warned that you are inside a PVP area:

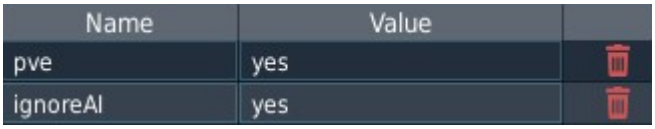


Blue should now again kill the red PVP plane. This kill has no consequences. After that, leave the PVP area. Once you cross the threshold to PvE country, you'll get a message



8.40.2.2 ME

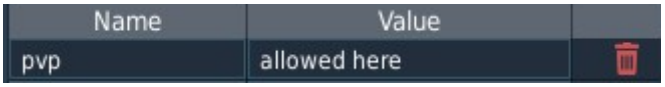
First, we'll inspect unGrief's config zone. It uses the standard set-up for a PvE mission:



- 'pve' = true makes all player-player kills illegal
- ignoreAI = true allows players to kill AI units of their own side without consequences, just like any 'normal' mission without unGrief

- Since no ‘warnings’ attribute is present, warnings default to ‘true’, giving all players notice when they cross from PvE to PvP and vice versa.

If you look at the map, there’s a big zone that uses map objects to highlight it red and label it as ‘PvP’ so everyone can clearly see where the ‘PvP’ arena is located. These markings are created using standard map objects.



More importantly, though, there’s also a trigger zone with a lone ‘pvp’ attribute. The mere presence of this attribute turns the zone into a PvP arena. The value ‘allowed here’ is ignored and can be anything. I chose that value to make it slightly clearer what this attribute is about

Since there is no ‘strict’ = true attribute present, for this zone, relaxed PvP rules are in effect: for a legal PvP kill only the player that is shot down must be inside the zone.

8.40.3 Discussion

Multiple PvP arenas
unGrief supports multiple PvP zones per map, and they can have different rules per zone. Be sure not to overlap PvP zones when they adhere to different rules, as in such cases, the particular rules chosen to judge a kill can be unpredictable. Try to add another PvP zone to the map

Not necessarily circular
PvP Zones don’t have to be circular; you can use Quad-based zones as well. This is especially useful if you want a PvP zone follow some terrain feature (e.g. coast)

Strict PVP Rules: an individual choice
Individual PvP zones can follow different rules with regards to kill strictness. This is set in the ‘strict’ attribute given per PvP zone. When the strict attribute is not given, or set to false, a PvP zone conforms to relaxed rules (only the killed plane must be inside the PvP zone at the point of kill).

8.41 Reinforcements a la Carte (Radio Menu, Cloner, Dynamic Reinforcements, Mission Restart)

8.41.1 Demonstration Goals

This mission primarily explores how to use radio menus to accomplish various goals:

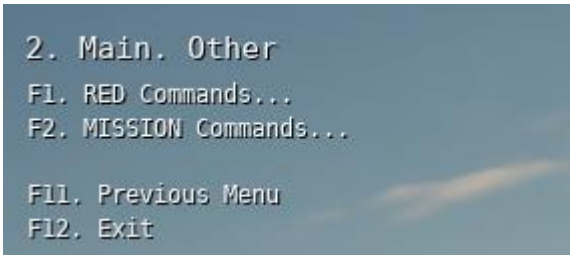
- Calling in reinforcements / spawning on command
- Coalition-specific menus
- Group-Specific Menus
- Global (all coalition) menus
- Restarting the mission via menu
- Cooldown messages for



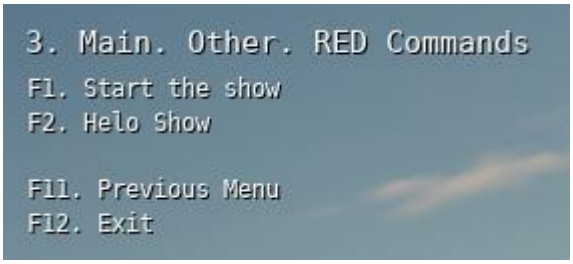
8.41.2 What To Explore

8.41.2.1 In Mission

Start the mission, choose the RED side, and enter RED Frog's cockpit. Take a look around at the beautiful rising sun, then choose Communication. Use 'Parent Menu' until you are in the 'Main' menu that offers 'F10 Other'. Choose it.



You are presented with two menu options: "RED Commands" and "MISSION Commands". Both are menus that are installed by DML based on Radio Menu zones. Note: the order in which those two commands appear may change (Mission commands comes before Red commands; this is how DCS menus work. The important thing is that the menus are present)

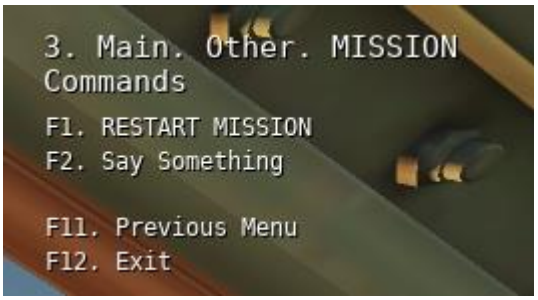


Choose “RED Commands…”, A new menu appears. Choose ‘Start the show”, and marvel at the group of Su-27 that spawn in, and then perform a flyby.

Call up the same menu and spawn another flight, and then another. Each time that you choose the “Start the show” item, a new flight is created.



Now choose ‘Helo Show’, and look to the left. A Mil-8 Hip spawns and takes off close to you. As fast as you can, choose ‘Helo Show’ again. If you are quick enough, instead of another Hip, a message like the one to the right appears. This menu item is on cooldown and can only be used again after the cooldown times out. The remaining time is displayed with the message.

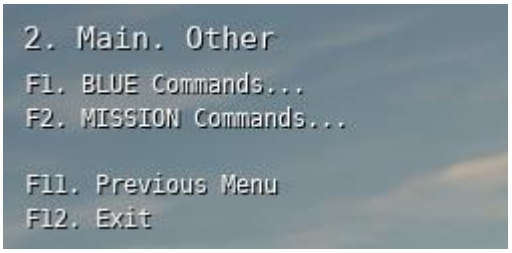


Now enter the ‘MISSION Commands…” menu and inspect the commands this menu allows.

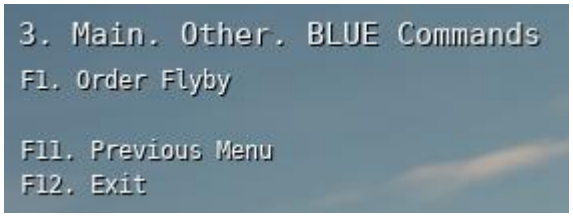
Choose ‘Say Something”, and note that a message “I say SOMETHING” is output to the screen

Then, choose “RESTART MISSION” and note that this (unsurprisingly) causes a re-load and re-start of the mission.

Now choose BLUE side and enter BLUE Frog’s cockpit. Again, navigate to the top menu, and choose “F10 Other”.

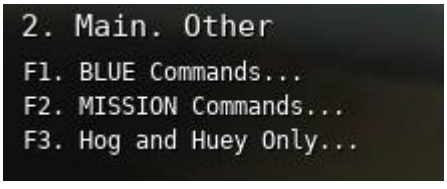


Note that the menus displayed this time are different: Now there’s a “BLUE Commands” menu where there previously was a “RED Commands” menu, while the MISSION Commands appears to be the same as what was offered to red.



Choose the BLUE Commands menu and inspect the ensuing options. Instead of red’s multiple commands, you are now only given a single command, “Order Flyby”. Choose it, and marvel at the four Tomcats performing a low flyby. As before, you can “order” multiple flights of Tomcats, one each time that you choose ‘Order Flyby’ from the menu.

Change slots to the blue “Double Augh” group (when you run this on a server, you will need the password “1234” to select the slot) and enter the cockpit. Now navigate to the “Other..” menu and observe that there now is an additional meu available: “O Sole Mio”. Choose it to receive a secret message.



Now enter the ‘Hogger’ (note: requires the A-10A module) or “Huey Blue” (note: requires UH-1H module) and call up the “Other…” menu. You’ll see a new menu item “Hog and Huey only”, while the Huey also shows an additional “Heli Only” menu.

Switch slots again to the Ka-50 “Karmov Blue” (Note: requires Ka-50 module). Taking a jaunt to the Other… menu reveals that here, in addition to “Blue Commands” and “Mission

Command” we have a “Heli Only…” menu, but no access to the “Hog and Huey Only” menu that we saw in the Huey before..

Finally, purely for entertainment, after quickly ordering a couple of flights of blue Tomcats, change both slot and side to red, then enter the Red Frogfoot, then order some flights of red Su-27. Now lean back and enjoy some aerial ballet as the groups engage each other.

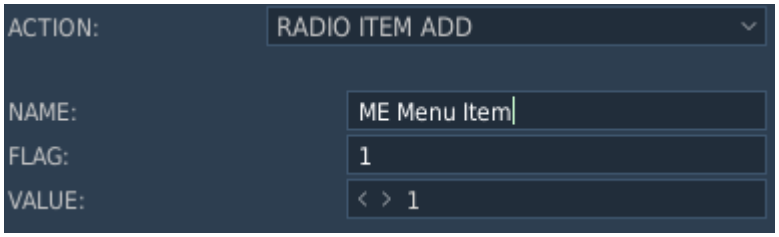
8.41.2.2 ME

This mission uses three zones for creating the various menus that you saw as player: one each for red and blue, and a common menu that is displayed to both.



The Red and Blue menus each have a ‘coalition’ attribute that controls their availability to players of each side, while the “All” menu omits a coalition attribute, and hence is available to all.

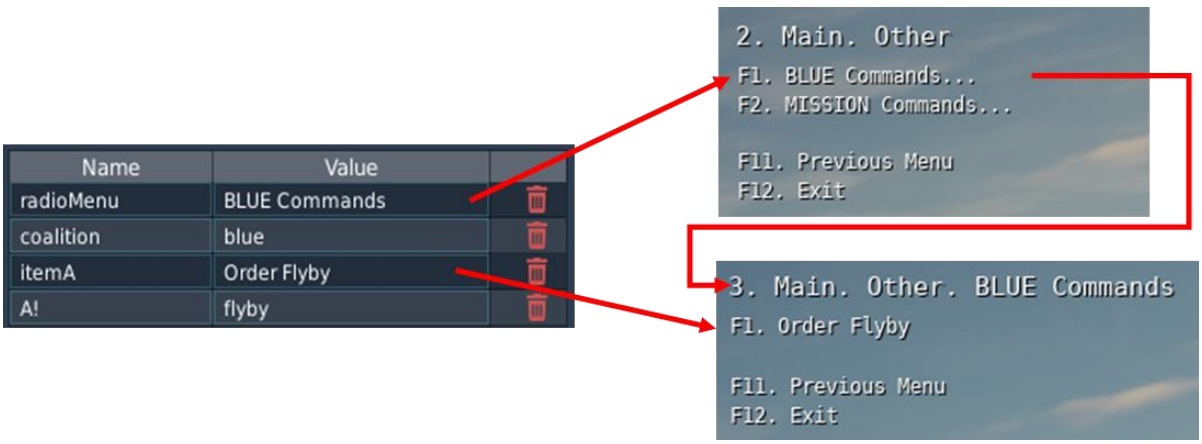
Similar to Mission Editor, DML’s radio menus do little more than setting flags, with a few important differences:



- In ME you directly add a menu *item* to the F10 Other menu, not a menu with its own sub-items.
- In ME, you set flags to a specific value, which usually means that you can only use this radio menu item once, and then need to re-set it before you can use it in a meaningful way again (perhaps by using a DML Radio Trigger module).

DML’s radio menus work slightly different:

They install their own menu entry in the F10-Other menu, which then can contain up to four separate items (with the attributes itemA, itemB, itemC, itemdD).



Additionally, when a player chooses one of the items, they use DML's flag methods to change the indicated flag(s), meaning that they can do more than simply set a flag to a specific value, and in DML terms can be readily used to issue multiple commands. Since it's so easy to re-issue the same command again with DML Radio Menus, it comes pre-installed with a cooldown option which we are going to explore soon.

8.41.3 Discussion

After gaining access to flags that we can control with menu items, the rest becomes. Simple:

SPAWNING ON-DEMAND

Actually, spawning (reinforcements) on-demand becomes trivial when combining a cloner with a radio menu: simply wire the menu items flag (e.g., flyBy from itemA! output) into the cloner's clone? input, and you are done. This is done for the Tomcats, Hip, and Flanker cloners.



COOLDOWN WITH COOL REMAINING TIME

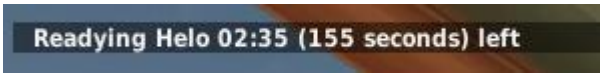
Since Radio Menus makes menu items so convenient and quick to use repeatedly, they often require a method to ensure that they are not used too quickly in succession. RadioMenu provides built-in cool-downs for items. Even more conveniently, when on cooldown, such a menu item can provide feedback that the item is currently under cooldown.

In this demo, the Red Side's 'Helo Show' menu item uses a 164 second cooldown before it can be used again.



If a player on the same side tries to re-issue that command before the cooldown times out, the 'busy' message is displayed.

Now, that 'busy' message has a very interesting built-in property: it can access and display the currently remaining cooldown time much like a messenger can interpret the value of a flag as a time value. Using wildcards such as <:s> and others, we can format the busy message to display the cooldown in a friendly, human-readable form.



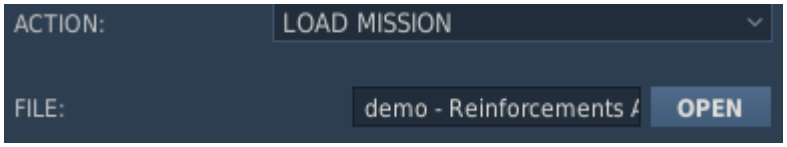
In the demo, the full busy string reads:

```
Readying Helo <:m>:<:s> (<s> seconds) left
```

The various <x> wildcards are replaced by their corresponding values.

RESTARTING A MISSION

Again, since we can selectively set a flag with radio menus, it is very easy to



restart a mission by wiring it into a ME trigger that has a LOAD MISSION action that points to itself, which will prompt a re-load and re-launch, even on most multiplayer servers.

Note that normally you want to guard against accidentally choosing this item, so put it in its own menu with only a single ‘restart’ item. That way, people who accidentally went into this menu can back out without aborting the current mission

Note also that restarting a mission is a dangerous, potentially grievous item to offer to random people on a public server, so be careful, and perhaps restrict access to menus (see below) to a few select players.

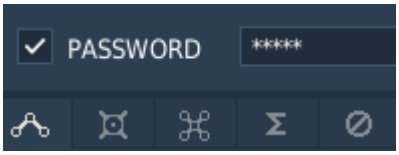
GROUP-ONLY MENUS – SECRETS “R” US

The group “Double Augh” has their own menu that is only visible to members of that group. For obvious reasons, you can only specify groups that contain player aircraft.

Name	Value
radioMenu	O Sole Mio
group	Double Augh
itemA	For your Eyes Only
A!	Secret

While this ability has many applications in normal missions, you can also use it to “protect” sensitive commands like restarting a server or saving a mission on multiplayer servers.

How so? DCS provides password-protect player slots (note: works only in multi-player). If you want to protect menus from being accessible only to some people, use that password feature to protect a slot, and assign the sensitive menus only to members of that group.



Be advised that this is merely a superficial deterrent for casual would-be griefers. The security provided by this mechanism will not deter a determined attacker.

TYPE-ONLY MENUS

Similar to missions that are only available to specific groups, you can make menus available to specific unit types, e.g. “UH-1H” or “A-10A”. Here we see two types listed: A-10A and UH-1H. We also see that coalition is set to blue, so all Hueys and A-10A Hogs (but not A-10C nor A-10C II) in the blue coalition will have access to this menu. That is why you saw this menu in the Huey and Hog, but not the Frogfoot on the blue side.

Name	Value
radioMenu	Hog and Huey Only
types	A-10A, UH-1H
coalition	blue
itemA	Hog and huey, sitting in a tre
A!	huig

When you jump into the RED Hog (A-10A), you see that there is no “Hog and Huey Only” menu – because although the Type matches (A-10A), the coalition for this unit does not.

Also, we have a menu that is available in all helicopters (you can jump into the

Name	Value
radioMenu	Heli Only
types	Helos
itemA	Helicoper menu is here
A!	helo

Karmov RED and verify that you also have access to this menu).

This is done with a “Helos” wildcard types attribute that automatically includes all helicopter types. Since there is no ‘coalition’ attribute for this menu, it is available to all sides (which is why the blue Huey, blue Karmov and red Karmov have access to this menu).

8.42 Delicate Subjects (‘brittle’ exploding units) tbc

8.42.1 Demonstration Goals

From time to time, you may want to use objects or units that simulate dangerous cargo, or must be protected at all cost. Delicates is a way to do just that – a unit or object designated as ‘delicate’ explodes after receiving even the slightest amount of damage. Placing multiple delicate objects / units close together can result in a chain reaction, so be mindful when placing them.



Note that due to its convenient side-mounted guns that work on the ground, this mission requires access to the UH-1 Huey module.

8.42.2 What To Explore

8.42.2.1 In Mission

Enter the Huey, and switch to the *right* gunner’s seat. Aim at the white civilian car, and quickly squeeze the trigger to shoot as few bullets as possible so that you merely graze the car.

If you do it right, the car will explode, slightly injuring the infantry next to it, causing a secondary explosion (that infantry is also ‘delicate’)

The infantry on the far left should not be affected.

Switch to the *left* gunner’s seat and take control of the mini gun. Take aim at the various objects, and note that all of them, with one notable exception, blow up as soon as you hit them with minigun fire.

The APC, on the other hand, remains impervious to all minigun fire, and even the explosions caused by the Hummer and other objects. Even though the APC is designated a ‘delicate’, it can’t be made to explode by hitting it with a minigun: the minigun does not penetrate its armor.

8.42.2.2 ME

There is very little surprise here. The most important take-away are



- as demonstrated by the two infantry, ‘delicate’ status can be assigned to individual units. Even though the group “Blown Away” consists of two infantry units, only the one inside the ‘delicate inf’ zone is ‘delicate’ and explodes when only slightly injured. The second unit which is outside the ‘delictae inf’ zone does not.

- You can assign delicate status to units, static objects and static objects that are cargo, but not scenery objects. Be advised that static objects may possess some unexpected properties, such as being impervious to small-arms fire.

Name	Value	
delicates		
power	1	

8.42.3 Discussion

Be careful when placing multiple delicate objects in close proximity, as they can easily cause a chain reaction.

Challenge:

Arrange objects in such a fashion that they create a chain reaction that lasts for at least 5 seconds.

8.43 Forever-looping Spawns (Server-friendly endless respawns)

8.43.1 Demonstration Goals

One of the many challenges for multiplayer-servers is providing a mission that can run 24/7 without the ‘background’ units running out of things to do. All units in a mission eventually run out of things to do: no more enemies to engage, no more routes to follow.



So how can you run a 24/7 server that is always fresh with units? One common practice is restarting your mission every few hours (this also keeps the impact of memory leaks at bay). But even with restarts every few hours, there often comes a point after a couple of hours where your main points of interest become points of boredom, nothing happens there anymore.

In this demo we explore a few simple design patterns that allow ‘always-fresh’ content.

8.43.2 What To Explore

8.43.2.1 In Mission

Enter the blue Frogfoot’s cockpit, and change to F10 Map view.

In front of you, an AH-1 is taking flight. A few seconds later, it disappears, only to re-appear on the ground, and take off again. This repeats over and over.

On your left, there is a lone redforce infantry. Should it be killed, it respawns immediately, as often as it is killed.

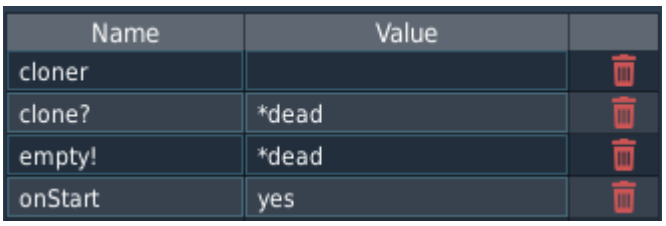
Then there are two ATGM Strykers. Observe as they trundle along their routes. One is driving down Senaki-Kolkhi’s 09 runway, the other one is cruising up Senaki’s tarmac in a NE heading.

Increase time scale to accelerate the action. After a while, the Stryker cruising up the tarmac suddenly disappears, and is replaced by a new Stryker driving back the opposite way. Once it reaches the end of the tarmac, closing the loop.

Once the Stryker driving down Kolkhi’s 09 reaches the end of the runway, it disappears and a new one appears at the beginning of the runway, starting anew.

8.43.2.2 ME
So, let's look at the most useful 'endless' design patterns:

RESPAWN ON DEATH



This is perhaps the simplest of all, and can be accomplished easily with nothing more than a single cloner, as shown with the perpetually resurrected infantry:

- Set onStart to true to have the group spawn when the mission begins
- When the group is destroyed, the cloner sends a local signal on empty!
- This is fed back into the clone? input, creating a feedback loop.

A big advantage of this design is that since all flags are local (start with an asterisk '*'), we can rapidly copy/paste this zone onto as many groups as we like.

TIMED REGULAR RESPAWNS



Another method to keep a mission fresh is to respawn the group regularly after a timer runs down. In DML this can be easily accomplished by stacking a pulser onto a cloner.

- The pulser provides a signal every n seconds (here randomized between 10 and 20 seconds; in a real mission this would usually a be a much higher value, say every hour = 3600 seconds)
- The signal (pulse!) is fed into the cloner's input to trigger the spawn. Since the pulser by default sends an initial pulse, the cloner does not require an onStart = true to clone the group at mission start.
- Every time the pulse's timer runs down, the previous clone is de-spawned and a new one is spawned.
- Note that if the group is destroyed, a new spawn is also immediately created (empty! also bangs the local restartHeli flag. Omit the empty! bang to have the group appear regularly, but keep it destroyed for a time (until the pulse timer runs out)

RESPAWN ON REACHING GOAL

A very good, albeit slightly more involved method to keep a mission in motion involves two trigger zones: a spawn point, and an end point. The idea is that when the spawned group reaches the end point, it de-spawns, and re-spawns at the starting point. Like the previous cases, we accomplish spawning with a cloner.

Name	Value	
cloner		
clone?	restartLoop	
empty!	restartLoop	
preWipe	yes	
onStart	yes	

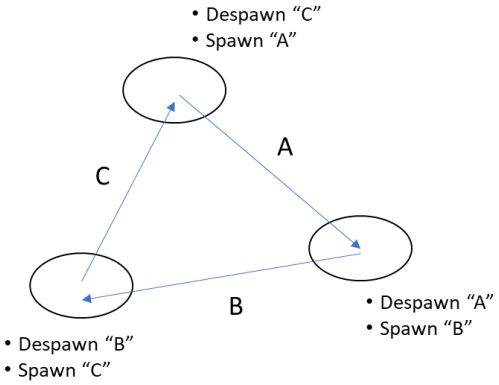
We detect that the group has reached the end point with a unit zone that creates the respawn signal for the cloner. There are a couple of noteworthy items here:

Name	Value	
unitZone		
lookFor	Loop*	
enterZone!	restartLoop	

- unitZone uses a wildcard “Loop*” for its lookFor attribute because a cloner generates groups with unique names based on the original (template) group’s name, and we want to detect all groups that are created by the cloner.
- This time we can’t use a local flag, because we need to communicate signals between trigger zones (the end point). This makes this kind of endless respawn less simple to copy and maintain.

CASCADED LOOP

A variant of the “Respawn On Reaching Goal” method is a cascade of spawns that ultimately form a loop:



- When a group reaches their end point, instead of re-spawning at the start (like in the example before), it despawns and a new group starts at that point
- The new group goes along its route until it reaches its own endpoint where it despawns and a new group is created in its place.
- This continues until finally, the last end point is reached which triggers spawning of the first group, closing the loop.
- Like the previous pattern, this one is not as easy to copy/paste because it requires zones that communicate with each other, and cannot solely rely on local flag.

Although cascaded loops require more effort to set up and are notoriously difficult to debug, they have some big advantages:

- You can break that part of the mission into smaller, more manageable tasks
- It’s more flexible: you can have destroyed groups revert back to the last point they reached or any other point in the cascade

- Gaming-wise, a cascaded loop is much more interesting: it mimics a group that heals/repairs and reloads over time: every time a group reaches the end of its particular leg, it is repaired, restocked and reinforced.

8.43.3 Discussion

Of course, you should experiment with above patterns, and fit them to your requirements. A couple of ideas that you should try to implement right now:

- *Combined Respawn Patterns*
  We already do that in the demo: when the group in the timed respawn is destroyed, it generates an immediate respawn. Similarly, the other patterns respawn on destructions to ensure that there are always living groups in the mission (the goal is to keep a mission entertaining 24/7 – on a server)

- *Delayed respawn*
  Currently, all spawners re-spawn immediately. A better version is stacking a delay flag module with the cloner, and run the respawn signal through a delay. Use local flags for the delay to increase portability

- *Message on respawn*
  Even better, also stack a messenger on the cloner to alert players that a group is about to respawn (and/or that a groups was destroyed)

- *Count kills and trigger events*
  Since each respawn signal has to be change in a flag (in DML at least), you can count the respawns, and do lots of interesting things with it (for example additionally spawning a boss group every Nth time)

- Bl*ock respawns with a gated switch*
  Use gated switches to break a loop and prevent respawns for some time. Make sure to also have a global timer running in that case so that even after a loop was broken, eventually it is restarted.

- *Menu driven (re-) spawns*
  Of course, you can always offer spawners that are triggered by radioMenu modules. We've dedicated a whole demo ("Reinforcements a la carte") for that, so this is just a reminder to allow people in your mission that option.

8.44 Impossible Impostors – Using Impostors

8.44.1 Demonstration Goals

This mission shows how to use impostors, and highlights their various options and when to use them. Impostors are incredibly useful, simple to use, and allow you a number of sneaky tricks in your missions.



So why are impostors so useful? A couple of reasons

- In impostor form, they use less CPU because no AI controls them
- You can ‘freeze’ groups until they are needed
- You can switch back and forth between impostor and AI controlled (with some restrictions), making them ‘animated on-demand’

So strap in, there’s a lot to explore in this mission.

8.44.2 What To Explore

8.44.2.1 In Mission

Jump into the Frog One Su-25T, hit F2 and take a look around. Take note of the following:

- The silver A-10 in front of you
- The green A-10 in front of you
- The AH-1 Cobra on your left
- The two mixed groups of vehicles and infantry slightly to the right of your aircraft
- The group of three Infantry on your four o’clock position

Now press F2 (aircraft view). Notice that currently, yours is the only AI-controlled aircraft in the arena: the F2 view does not jump to other aircraft. The two Hogs in front of you are static objects.

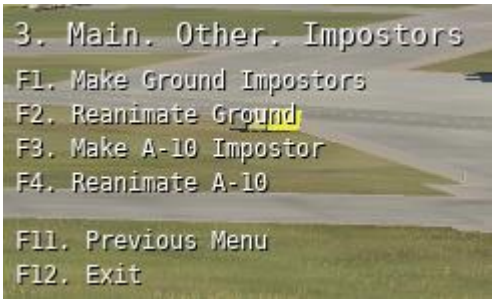
Now press F7 (Ground Unit View). You'll be able to switch between two ground units:A red one that is busy attacking a blue unit that does not fight back (or is already dead), and a blue unit that stands close to its twin. Note that you can only switch the view between these two ground units – all other ground units are static objects. Switch to the red Infantry. Note that

- It attacks, and kills the blue infantry static object
- A message appears:
  

- This proves that Ai attacks static objects
- Static objects can be easily killed and don't shoot back

Press F2 again to return to the outside view of your Frogfoot.

Now choose Communication→ Other→ Impostors→ Reanimate Ground.



Look around. Note that

- A lot of labels have appeared on a multitude of ground vehicles and infantry. Press F7 to step through them. You now have many units to step through
- The units look exactly like the static objects that occupied their positions just a few moments before.
- Note that some units have winter, while others have desert, and most use default liveries. Yes, Impostors has full livery support.
- Your RWR tone activates because the patriot site behind you has come on-line
- There was no 'blinking' visual artifact (you'll see the difference when we reanimate the Hogs).
- The two groups in front and to the right of your aircraft start moving, marshalling on their way to their first waypoint
- A message from a tracker module has appeared:
  

- When some of the units in front of you have moved a couple of yards, choose Communication→ Other→ Impostors→ Make Ground Impostors. Note that all units suddenly freeze in place, and the RWR tone stops. All units have turned into Impostors (static objects) again. Choose Communication→ Other→ Impostors→ Reanimate Ground to again turn them into AI-controlled units and continue on their way.

Wait until the two groups of infantry have cleared the immediate area in front of you (might take a minute). Look at the two static A.10 in front of you: the silver and green one. Now choose Communication→ Other→ Impostors→ Reanimate A-10. Note that

- The silver Hog seemingly vanishes
- The green hog "blinks" (briefly vanishes and re-appears) and starts to move a few seconds later (when no vehicles obstruct the path)

- The Cobra helicopter blinks, then takes off and starts flying to the east
- Use F7 to step through all controlled planes. Rediscover the silver A10 in flight above the airfield. It's alive and well, in the air.

Step until you again see the green A-10, and wait until it has taxied a few seconds and turned some 180° to head for the runway. Choose Communication→ Other→ Impostors→Make A-10 Impostor

Note that

- the A-10 on the ground stops in its tracks and has turned into a static object.
- The Cobra helicopter also drops to the ground and rests there completely undamaged
- The A-10 that was airborne now sits somewhere on the ground to the northwest (where it was flying just before), completely unharmed

Now choose Communication→ Other→ Impostors→ Reanimate A-10 and note that

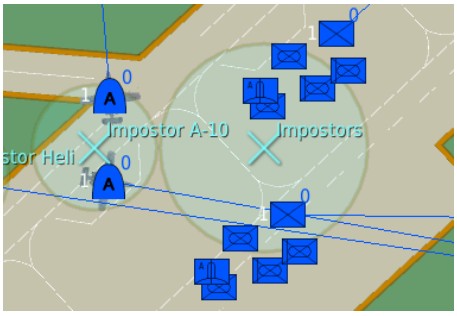
- The airborne A-10 is back in the air and is heading for waypoint 1
- The Cobra takes off where it dropped and flies towards waypoint 1
- the ground A-10 teleports back to the slot in front of the Frogfoot. Why? Because DCS is mysterious.

For the remainder of this demo you need access to a UH-1 module. This is merely because it has very convenient side-mounted miniguns that we can use to manually kill some units/objects. Replace the Huey with your favorite method to selectively destroy units.

Restart the mission, enter the Huey, and switch to the left side gunner (press '3'). Use the minigun to destroy one of the three static infantry. Now choose Communication→ Other→ Impostors→ Reanimate Ground. Note that the group now only consists of the two surviving soldiers. Destroy one of the two remaining soldiers, and then choose Communication→ Other→ Impostors→ Make Ground Impostors. This only turns the lone surviving soldier into a static object

8.44.2.2 ME
So let's look at the various use cases one by one.

MULTIPLE GROUPS PER ZONE
Just like many other modules, any group that has at least one unit inside the Impostor Zone will gain the impostor ability. In the example to the right, each zone manages two groups, even though not all units reside inside the zone.



Remember that Impostors always works on entire groups, never individual units.

THE BASIC IMPOSTOR
The simplest fully controlled Impostor (and that is used extensively in the mission is one with watchflags for switching between the two states, and optionally the onStart attribute. You'll find

Name	Value
impostor?	ggoImp
reanimate?	goGroup
onStart	yes

yourself using onStart for many groups that simply await activation when something relevant happens as decided by your mission.

IMPOSTOR FOR AIRCRAFT ON THE GROUND

Since AI aircraft on the ground spawn with some DCS-imposed, mandatory sanity checks, they always spawn on un-occupied slots. This means that there must be a short interval between removing the static object that occupies the slot (and conveniently prevents other aircraft from spawning there).

Name	Value
impostor?	hgoImp
reanimate?	goHawg
onStart	yes
blink	0.1

Blink values from 0.1 to 0.2 are usually sufficient.

USING IMPOSTORS WITH A GROUP TRACKER

Impostor zones have built-in support for DML group trackers. Integration works by invoking addGroup() at the tracker when the Impostors are turned into AI-controlled units and removeGroup() when they are turned into impostors.

You can see this in action when the mission starts, as the tracker is using zone-local verbosity. When the mission starts up, it shows the warning that

```
+++gTrk: Note - group <Three Amigos> wasn't tracked by <The Tracker>
```

This warning comes up because the Impostor Zone turns the tracked group “Three Amigos” into imposters (due to the onStart flag), notices that they are tracked by a tracker, and accordingly tells the tracker to remove this group. Since it wasn’t yet registered with the tracker, and verbosity is active for that zone, it shows the warning.

Name	Value
impostor?	ggoImp
reanimate?	goGroup
trackWith:	The Tracker

To use a group tracker, simply list its zone name in the “trackWith:” attribute

LINKING ZONES TO UNITS WITH IMPOSTERS

DML supports linking a zone’s position to a unit with the “linkedUnit” attribute. Impostors respect this link and preserve it when a unit is turned into an Impostor, and return it to the unit when it is turned back into an AI-controlled unit.

Name	Value
linkedUnit	Amigo One
verbose	yes

If you add the ‘verbose’ flag to the zone that is linked to an impostor, you will get a message each time a zone is linked to, and unlinked from an impostor like the following:

```
+++ipst: imp-linked zone <Linked to amigo one> to imp <Three Amigos-8200105>
```

8.44.3 Discussion

Here are some more subtle points that you may find interesting

- The two A-10 and a number of the vehicles use different liveries that carry over to their impostor version
- When units are turned back to their AI-controlled version, they usually start heading towards their first (not initial) waypoint from wherever they left off – unless they are aircraft that start from slots and use a different placement logic.
- You can use hot start for aircraft to ‘scramble’ jets seconds after reanimating them
- If you intend to repeatedly stop ground units by turning them into impostors and later continue on their way, make sure that they only have two waypoints: the initial point where they spawn, and waypoint one

Challenges:

Here are some fun things to try with impostors:

- Make the A-10s (and helicopter) activate when the red infantry kills the blue impostor
- Add an impostor SAM that reanimates when the airborne (silver) A-10 gets close
- Make all ground units become impostors when your Frogfoot is more than 2 miles away from Senaki-Kolkhi
- Make a group of vehicles turn into impostors when they arrive at their destination.
- Do the same with an group of aircraft
- Create an escort mission where whenever the player is too far away from the vehicle they escort, the vehicles stop (turn into impostors), and continue when they are close enough. Hint: use a zone that is linked to an impostor.

8.45 Being Persistent (load and save a mission)

8.45.1 Demonstration Goals

This mission demonstrates basic mission persistence: loading and saving ME-placed ground units and static objects, as well as saving and loading flags (or rather, their values)

This mission also demonstrates “manual” saving and how to force a ‘fresh start’

In order to be able to run this mission, your DCS installation must have performed the required steps to un-sanitize “io” and “lfs”. Please refer to the “Persistence” section on how to do that.

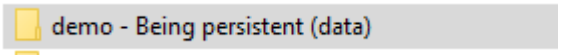
NOTE:
This mission will create a folder called “demo – Being persistent (demo)” in your Missions directory, and save all mission data into that folder

8.45.2 What To Explore

8.45.2.1 Before Starting the mission

Open your missions folder in Explorer and keep it open. When the mission begins, you will see a folder “demo – Being persistent (demo)” appear when the mission starts up for the first time and has read/write access. If you already have such a folder in your Missions directory, please delete it.

Start the mission and keep and eye on the Missions folder. You should see a new folder appear:

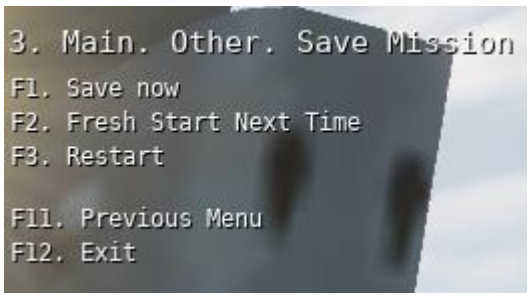


Open the folder, it should be empty.

Now switch to DCS, in-mission

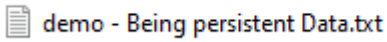
8.45.2.2 In Mission

Choose the Frogfoot, and enter the cockpit. Observe the various units moving about. Wait a few seconds until at least some of the infantry or vehicle has passed your plane (so you can tell for sure that when you load the mission again, the units are restored to other positions than their start position.



Choose Communications→Other→Save Mission→Save Now

A new file appears inside the newly created folder:



Note: you can open this file with any text editor. If you change data inside (it's JSON format), be sure that you know what you are doing.

Choose Communications→Other→Save Mission→Restart

A second or so later, the mission will restart. Choose the Frogfoot again, and once you are inside the cockpit, look outside. The mission should have loaded, and the infantry that last time started to the right of you should now start at the point they were when you made your save.



For the next test, change into the UH-1, and switch to the right-side gunner (4). There are four infantry standing next to your helicopter. Destroy the closest one, and then one of the three remaining. The other two will start running for cover. Save the mission and restart.

Jump back into the Huey and take the right-side's gunner position. Note that the two dead units are now removed, and the remaining two are no longer running for cover, but are standing at the location the were at when you saved the mission. You may have to inspect the F10 map if they got too far.

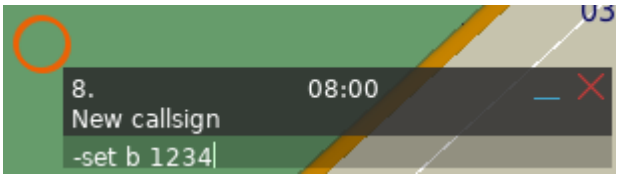
Now switch to the left side gunner (3). Take the minigun and destroy the Hummer and oil tanks (they are static objects). Save the mission, restart, and return to the left side gunner.





Note that now, the hummer shows up as destroyed, but the oil tanks, although they were clearly dead, look unblemished. This is a peculiarity in DCS: some static objects look OK even though they spawn as dead (oil tanks), while other correctly spawn dead.

Now change to F10 map view, and use the debugger's interface to change flag b to 1234 (place a map mark, and enter "-set b 1234", then click outside the mark's text field).



The debugger responds with



Save and then restart the mission. Use the debugger to recall the value of flag b (with '-show b')



And finally, choose Communications→Other→Save Mission→Fresh Start Next Time. You'll see the following message:



Re-start the mission. This time, it starts fresh, with all units and static objects restored to original starting positions and alive.

8.45.2.3 ME

There's very little to see here, most of the ability is provided by the persistence and unitPersistence modules transparently.

The most interesting part here is persistence's config zone. We see that persistence should save the mission (saveMission?) whenever a change is detected on the 'save' flag and restart fresh next (cleanRestart?) time it observes a change on the 'clean' flag. Obviously, these are fed directly from the RadioMenu zone.

Name	Value
saveMission?	save
cleanRestart?	clean
saveFlags	a, b, c

Also note the 'saveFlags' line that we stacked into this zone that marks the flags 'a', 'b' and 'c' as flags whose values it should save and load with the mission.

Name	Value
radioMenu	Save Mission
itemA	Save now
A!	save
itemB	Fresh Start Next Time
B!	clean

8.45.3 Discussion

Remaining Fresh

Once you tell a mission to take a fresh start, it will do so. You may have noticed that there still is a mission data file. Don't worry, that data remains ignored every restart until you overwrite that old file with a new save. This is because missions cannot remove files from a directory unless they also have 'os' un-sanitized, something you should not do, nor request from those who run your missions.

Deleting the data file or mission data directory

You are free to delete the data file out of the data directory or data directory altogether any time you are not flying the mission. This will also cause a fresh start next time you run the mission. The 'cleanRestart?' flag is for convenience especially for dedicated server owners, who may not be able to easily access their server's missions directory.

Restricting access to saveMenu? and cleanRestart?

When you add a radio menu to give access to commands like save and clean, every player can access them. Are there ways to give access to these commands only to select people?

Yes – there are multiple ways to do so:

- Via one-sided menus. This only works if players can fly only planes from one side. You can then set up a password-protected aircraft on the other side, and set a RadioMenu with access to “saveMission?” and “cleanRestart?” that is restricted only to that side
- Or you can leave the debugger in the game (it consumes next to no resources), and use it to “-set” the flags that trigger “saveMission?” or “cleanRestart?” directly. Only those people who know which flags to set can then trigger a save or clean.

8.46 Sequencing Fun (Sequencers)

8.46.1 Demonstration Goals

Everything is a sequence. Sometimes, though, we must exert some control that what happens in our mission happens in the correct sequence. The sequencer module helps you keep the order of events in order, and this demo shows just a glimpse of what a sequencer can do for you.

Note:
To better appreciate how this demo accomplishes its feats, it helps if you are also familiar with the 'Impostor' module



8.46.2 What To Explore

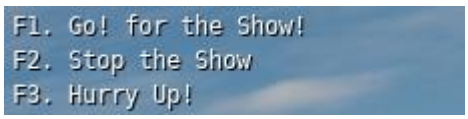
8.46.2.1 In Mission

Enter the mission in the Frogfoot, and simply wait. You'll see three warning messages appear one after the other:



Don't worry, there will be no restart- This is merely a demo of how you can sequence a number of messages, with an arbitrary time interval, and at the end trigger some action.

Now go to Communications, and choose Other→Start the Show. You have three options:

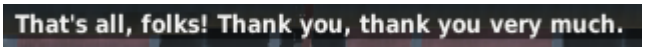


Let’s briefly discuss what these menu items do:

- “Go! for the show” starts and un-pauses the sequencer
- “Stop the Show” pauses the sequencer
- “Hurry Up!” advances the sequencer

Now start the show. One after the other, the following happens:

- The infantry soldier starts walking
- The AH-1 Cobra wakes up and takes off immediately
- The refueller pulls away from the Hog
- The Hummer pulls away from the Hog
- The Firetruck pulls away from the Hog
- The Hog wakes up and starts taxiing
- A message displays that concludes the ‘show’

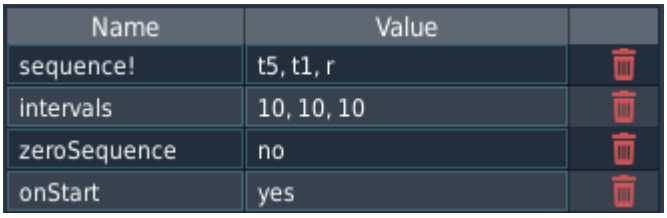


While all this happens you can use ‘Stop the Show’ to prevent the next stage from happening, and then use ‘Start the Show’ to continue.

While the show is running (but not when it is paused), you can force the next stage to happen immediately by using “Hurry Up”

8.46.2.2 ME
So let’s start with the messages that simulate a server’s shutdown warnings: In a real mission, your setup is likely slightly different: you’d space the stages by more time than we do here, and the final message would obviously also restart the server (which we are not doing here)

The sequence of flags that should be sent a signal are t5, t1, and r (as defined by the ‘sequence!’ attribute), with a wait period of 10 seconds before t5, 10 seconds before t1, and 10 seconds before the final flag signal (intervals).



Name	Value	
sequence!	t5, t1, r	
intervals	10, 10, 10	
zeroSequence	no	
onStart	yes	

Since we want the sequence to operate in a Wait-Signal fashion (first wait, then send the signal), the ‘zeroSequence’ attribute is false, which causes the seqencer to wait before it signals the first flag t5, and we start the sequencer when the mission starts up (onStart is true)

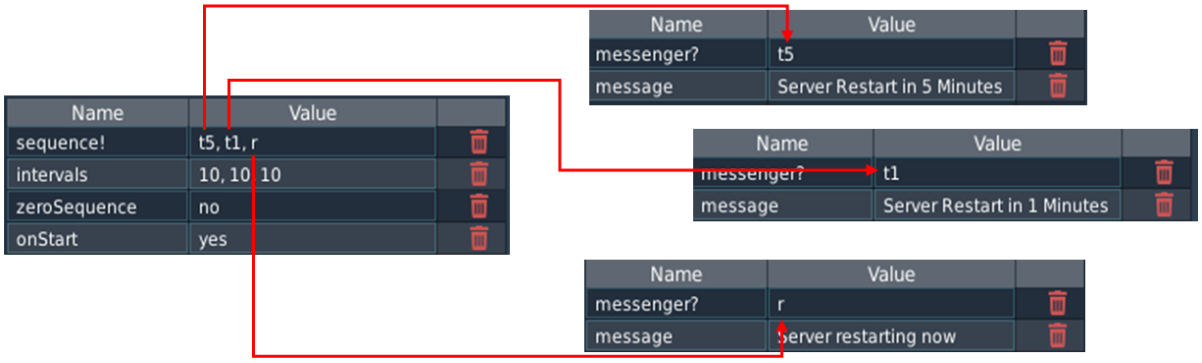
The flags t5, t1 and r are all connected to the inputs of their respective messenger modules, that are all set up similarly: they wait for a change on their input messenger? and, once received, proceed to output their message.



Name	Value	
messenger?	t5	
message	Server Restart in 5 Minutes	

Above outlines the basic flow for sequencers: a central sequencer connects to various modules via flags, and the ‘sequence!’ attribute determines the order in which flags are

signaled. The ‘intervals’ values determine how long the sequencer should wait between sending signals.



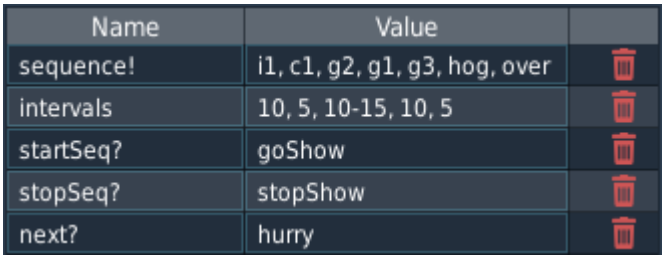
That schema should cover more than half of your sequencing needs: simulated dialogues, staged sequences for better immersion, regular events, or fixed-timed events in your mission are all covered.

But a sequencer can do far more:

- It can randomize the interval in a tightly controlled fashion
- It can immediately end the current waiting period and proceed to the next stage through an input signal, which is very helpful for building a common mission pattern: “A happens when the player does this OR the time runs out, whichever comes first”

The staged sequence of various vehicles pulling away from the hog demonstrates both:

Here, the sequencer is set up like before, and it simply uses a few more inputs to better control the sequencer:



- startSeq? looks for a signal on the flag “goShow” to start or un-pause (continue) the sequencer
- stopSeq? looks for a signal on the flag “stopShow” to pause the sequencer if it’s running
- next? looks for a signal on flag “hurry” to skip the wait and immediately go to the next stage in the sequencer

There are two interesting points here:

- “intervals” third interval reads “10-15”. This is a range, and when the sequencer encounters it, picks a random value in this range. That way you can introduce an element of random into the timing of the stages, without upsetting the order.
- A signal on “next?” causes the sequencer to advance to the next stage, overring the timer. This means that there are always two things that can cause a sequencer to advance to the next stage: a signal, and a time-out, and it always uses whatever comes first

8.46.3 Discussion

Pausing a Sequence does not pause the game
Keep in mind that a sequencer merely orders the sequence in which it sends signals on flags. This means that if you pause a sequencer, that will not pause the actions it may have already triggered. When you pause the Show after the infantry soldier has started moving, that soldier itself does not start moving.

About that default 24 hour interval
You may have stumbled on a peculiar default for the ‘interval’ attribute: it defaults to 86’400 seconds, which is 24 hours. Why is that?

Before I answer let’s briefly come back to the way how sequencers work: it advances to the next stage whenever the interval timer runs out, or a signal is received on the next? input, whichever comes first.

Now imagine you want a sequence that is entirely triggered by signals on next? – how would you do that? Simple, you may think, simply omit the ‘interval’ attribute. That would be correct, but since a sequencer always looks for an interval time, it uses a default. If we use a 1 second default, the sequencer runs through all stages in that many seconds. The solution is to default the interval to an arbitrarily large amount, and 24 hours is a nice compromise, as that is also the average amount of time most server run missions before they re-start them for better performance. If you need longer intervals, you must set the interval attribute accordingly.

8.47 Departures and Landings.miz (LZ)



8.47.1 Demonstration Goals

LZ is quite flexible when it comes to choosing which aircraft landing in or departing from the zone it is attached to, and with LZ it is very easy to trigger other modules or scripts when this happens. This demos shows how to create flag events when

- any aircraft takes off
- a player aircraft takes off or lands
- a unit with a specific name lands
- a group with a specific name lands
- an aircraft of a specific type lands

inside a zone, and we'll also demonstrate how this can be easily achieved with moving platforms like ships.

8.47.2 What To Explore

Note I:
There are some aspects of the mission that you can try interactively yourself, but this requires that you have access to a helicopter mod, and know how to fly it well enough to

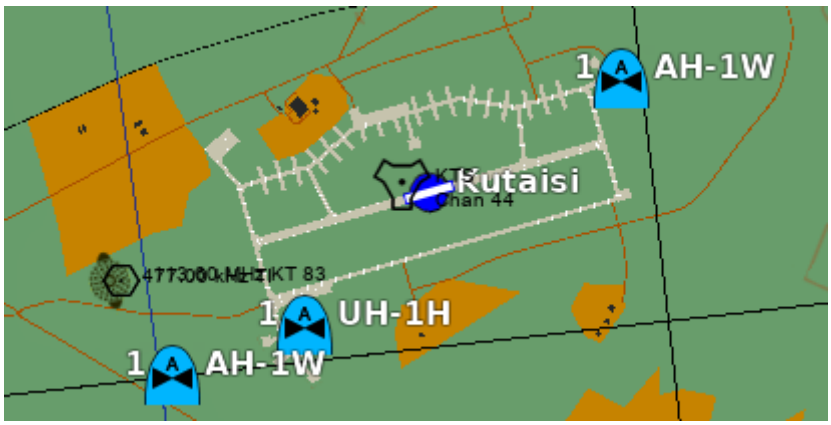
land on small platforms. It's not required though, because AI will eventually do all that for you as well

Note II:
You may find it helpful to use some time compression (CTRL-Y) if you are easily bored.

8.47.2.1 In Mission
Start the mission, and enter the "Frogger" Frogfoot. Press F2 twice to watch an A-10 starting it's run down Kolkhi's 09 and eventually taking off. A short while after becoming airborne, a message appears:



Switch to F10 Map view and pan over to the east until you are looking at Kutaisi. Note the three helicopters



Two of them will soon land, while the one in the upper right corner is gearing up for departure (a cold start, takes a couple of minutes). Each time a helicopter lands, you'll see one or more messages appear:



Then you'll see an A-10 arrive at Kutaisi, followed by



And the AH-1W departing from Kutaisi north-East:



If you don't have a Hip or Huey module, or don't' want to fly a gas platform approach:

Press F2 until you can observe the Mi-8 "Hip" getting ready to depart the Tarawa.



Watch it fly across to the gas platform and land on it



If you have a Huey or Hip module
you can fly the Hip from the Tarawa to the gas platform and back - and trigger the messages yourself. You can do the same with the Huey, and additionally, you can take off from and land on the small platform in the grass in Senaki-Kolkhi

8.47.2.2 ME
From a mission design perspective, this one is a cakewalk: the LZ generate one or two flag events that are wired into messengers. The only interesting thing here are the different set-ups for the LZ, and how they filter for aircraft that trigger.

The basic setup of an LZ that detects every landing and take-off inside its zone is depicted on the right: the events are triggered for any aircraft that lands or starts in the zone. There is not filtering for types, units, groups, players nor coalition, so any plane taking off or landing in this zone will create flag events.

Name	Value
LZ	
landed!	lK
departed!	dK

If you want to filter out a class of aircraft (e.g. only want to detect landings or take-offs from Helicopters), you can use the types attribute with the special 'Helo' type. This will ignore all fixed-wing landings.

Name	Value
LZ	
landed!	lKuHeloS
types	helo

Likewise, you may be interested only in landings or take-offs from players. This is controlled by the 'playerOnly' attribute. When set, all AI landings and take-offs are ignored.. In the example to the right, all take-offs and landings from any player-controlled aircraft (any coalition, any type, helicopter and fixed-wing) creates an output

Name	Value
LZ	
landed!	lKP
departed!	dKP
playeronly	yes

Instead of types, we can also filter for group names or unit names. When you filter for group or unit names, you can list multiple names, and the names can end in a wildcard '* meaning that only the part before the'*' must match exactly. The example on the right will create a landing or take-off event for any unit that has a name that starts with "Co". Remember that name matching for groups and units is case **in**sensitive.

Name	Value
LZ	
departed!	dKuHelo
units	Co*

And here we have an LZ that only reacts to landings by one side:

Name	Value
LZ	
landed!	lKuRW
coalition	blue

‘coalition’ is set to blue, so this LZ ignores all landings by red aircraft.

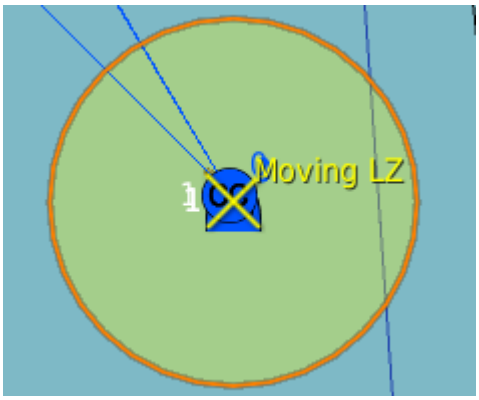
If you set coalition to 0 or ‘neutral’, the LZ will react to *all* landings, neutral, red and blue.

And finally, an LZ that is attached to a unit – here the Tarawa. This allows you to detect take-offs and landings that happens on moving units like ships and carriers, but may also be used with convoys, trains or other creative ideas.

Name	Value
linkedUnit	Tarawa
LZ	
departed!	dTa
landed!	lTa

8.47.3 Discussion

If you look at moving the trigger zone for the LZ over Tarawa, you’ll notice that it is disproportionally large. This is for a reason and important because there is a risk that it might not detect all landings or take-offs:



For performance, all modules in DML, like most DCS processes, update on a schedule, usually once per second. This means that a moving zone is also updated once per second. If the Tarawa moves during this time and the LZ is too small, there is a chance that the part of the deck that an aircraft that touches down or takes off from is outside the LZ – and the take-off or landing will not register (correctly, as they happen outside the zone)

So if you attach an LZ to a moving unit, allow for some ‘drift’, controlled by the maximum speed of the unit.

8.48 Willie Nillie (Willie Pete / Player Score drop-in)

8.48.1 Demonstration Goals

This demo shows how to use williePete and demonstrates PlayerScore's integration with williePete.

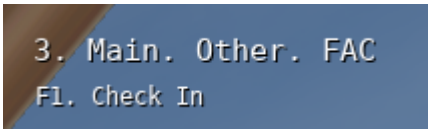


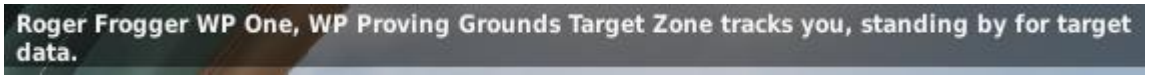
8.48.2 What To Explore

8.48.2.1 In Mission

Start the mission and choose the airframe that you prefer (and know how to operate sufficiently to place a wp rocket close to a target). In this documentation, we are flying the (free) Frogfoot, but you are free to choose any of the other airframes that the mission provides. They are all armed with WP munitions.

As soon as you enter the cockpit, go to Communication→Other→FAC and choose 'Check-In'. You should receive an acknowledgement that you are checked into WP Proving Grounds, and that it is tracking you. This means that whenever you fire a wp rocket into the ground, that the artillery for this zone will use that a target.





Continue your ingress towards the red forces arrayed on the airfield, choose air-to-ground, and arm the WP rockets. Place a wp rocket on the ground close to a group of red armor. Be advised that these troops have some teeth and will fire at you, so don't be too casual about placing the target marker. If you are new to firing rockets, your accuracy may be lacking, and you may require a few passes until your rockets fall true. Make sure that you review the rocket delivery procedure for your aircraft; the Frogfoot's rocket delivery system is quite accurate and easy to learn, so you may want to try it yourself.



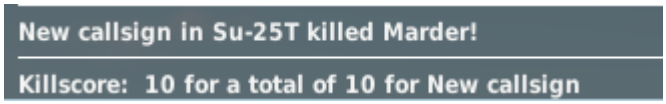
Once you have successfully placed a smoke close to your intended target, choose communications→Other→FAC→Target Marked…

If you can't find the "Target Marked" menu, make sure that you are signed into the target zone, and that the WP rocket hit inside the WP Target zone.

You'll transmit target coordinates and a few seconds later you should receive an acknowledgement. Make sure to maneuver out of the danger zone, because within 20 seconds, shells start raining on the hapless vehicles.



When the shells start exploding around your mark, some will hot close enough to a vehicle to cause damage and even outright destroy a couple. In that case you will see a message similar to this:



If the marker looks to be very close to the target, and the artillery barrage did not kill all vehicles, you can either try to fire another marker even closer to the targets, or simply tell the artillery to fire again: cool-down in this demo is very short, short enough to allow you multiple barrages on the same WP mark (note that by default, cooldown for a WP zone is too long to allow a second barrage on the same mark)

Once you have destroyed all red targets (including the tanks which require precise positioning and some luck with the shells landing very close or on top of a tank), all targets re-spawn.

8.48.2.2 ME

Setting up WP zones is straightforward: place the zone, add the wpTarget attribute and you can go wild with WP: everything's in place, including radio menus. That being said, there are some interesting nuances to WP:

WP Set-Up

This demo's WP setup is more geared towards fun and action, and less towards precision. Let's briefly walk through this Zone's set-up:



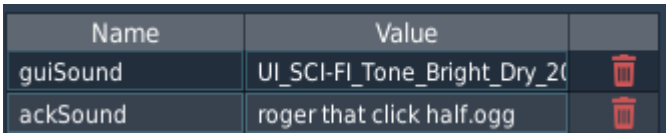
- It's a blue zone (meaning red planes cannot trigger artillery), verbosity is on (which you should turn off in production missions).
- *Cooldown* is set to a mere 20 seconds, allowing players to fire again almost immediately after the shells start hitting (making an unrealistically short re-load time).
- *Shell Strength* is set to 1900 making them quite lethal and destructive, strong enough to level buildings (you can try this yourself by firing a WP into some of the nearby buildings and tell the artillery to fire), and
- *Base Accuracy* is set to 70, making the shells hit inside wide a circle 140m (430 ft) in diameter.

- Finally, a *shellNum* count of 17 should allow for a good spread of destruction inside the target circle.

The high shell strength, combined with high shell count, and wide target circle allow for plenty and easy destruction close to the target marker. Since players don't have to wait long for the artillery to reload, they can re-try almost immediately, so that combination is ideal for training missions.

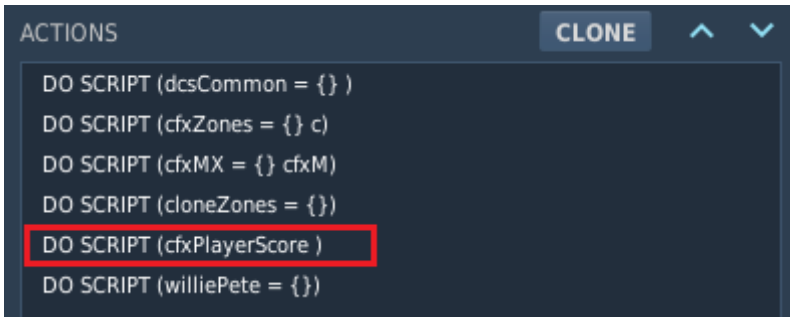
If you want to raise mission difficulty, lower shellNum, shellStrength and base accuracy.

Aural Feedback
There is a yellow wpConfig zone that provides some additional polish to this mission:

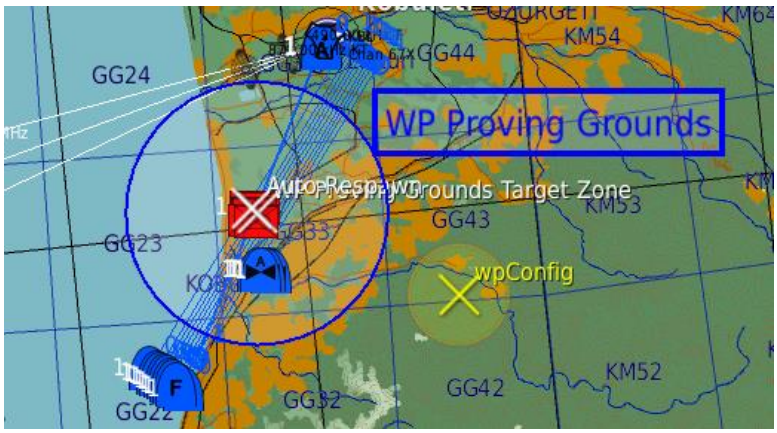


They provide interface audio effects to the radio menu interface with the artillery. Note that you need to add these audio effects to your mission manually (see the 'Load Audio' mission trigger).

PlayerScore integration
Finally, note that in this mission there is no zone evident that configures or controls PlayerScore, yet we get noticed when the artillery barrage destroys a vehicle. This serves to demonstrate the true drop-in nature of PlayerScore: all you need to do is add that module to your mission, and PlayerScore and WilliePete work together. Note also that there is no enforced order for WilliePete and PlayerScore – the dependency is run-time, not load-time so either one can load first.



Marking the Zone
Remember that Trigger Zones aren't visible on the F10 map to players, so it makes sense to mark them in ME for better playability. In this mission I have used a "Disk" polygon to mark out the Zone, and a Text Box to label it.



In missions with WP zones this is helpful for pilots: they can fence in and sign up on ingress, know which WP zone they approach (if your map has multiple zones), and they know when they are inside the zone and can commence marking targets.

Auto-Respawn

Name	Value	
cloner		
clone?	*respawn	
empty!	*respawn	
onStart	yes	

This mission makes use of a useful and common mission design pattern: endlessly respawning units. Once all red units are destroyed, the entire group automatically respawns. This is done by simply wiring the *empty!* output (which fires when the last unit of the last spawn is destroyed) into the *spawn?* input. We use a local flag (*respawn) as to not pollute the flag space and make it easy to copy/paste the cloner.

8.48.3 Discussion

Different voices?

WP Zones support the ability to play individual sounds as responses to radio command. This is so you can give each target zone a touch of individualism – but it of course requires that you provide your own audio for each. By default, the sound WP zones play is the same one that you define in the wpConfig zone, but each WP zone can override that sound with their own.

Multi-Unit Player Group

The group xxx has two player units in the same group. Note the WARNING that the mission gives at start-up. Using multi-unit player groups is not recommended with WP and can lead to unpredictable results.

Challenge: classic FAC mission



Let's try and create a simple, yet interesting mission with just the modules that we are currently using in the "Willie Nillie" mission.

Have the DML randomly distribute/hide enemy FARP tents over the map, and then task the player hunt them down and destroy them (which effectively mimics FAC missions from the 60s and 70s.

Hint: you can use the cloner's rndLoc attribute to great effect here.

8.49 Feats and autoCSAR

8.49.1 Demonstration Goals

This demo servers to demonstrate how the “Feats” part for Player Score works in conjunction with completing CSAR missions. It also shows how you use autoCSAR to automatically generate CSAR missions for any AI pilot who survives an ejection.



Note that to play this mission yourself, it is required that you have access to a transport helicopter module (Huey, Hip or Hind) and can fly it well enough to perform a CSAR mission.

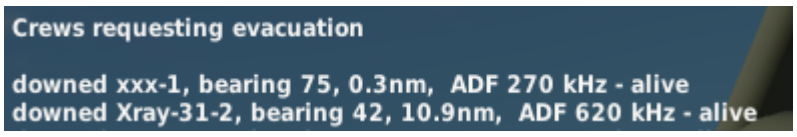
8.49.2 What To Explore

8.49.2.1 In Mission

Enter one of the three troop transport helicopters (Hip, Huey, Hind), switch to the F10 Map and zoom out until you can observe the flock of F-15 enroute to Kutaisi. Near Kutaisi, there is an SA-10 that starts shooting as soon as the mission begins. Some of the F-15 will be shot down, and some of the pilots will survive the ejection. Once one does and safely reaches the ground, a message comes up:



Go to Communications→Other→CSAR Missions and choose “List Active CSAR Requests” and you’ll receive a list similar to



Note that there is always one CSAR mission (XXX-1) that is very close to you. I put this one in to save time when we go about recovering the pilot (it’s just a short hop away at the airport). Look at the map and see if you can locate the downed pilot. Note the parachute on the ground next to the pilot, which shows it’s indeed a dynamically generated CSAR mission (if you switch to the downed pilot very recently after they touched down, you’ll notice that

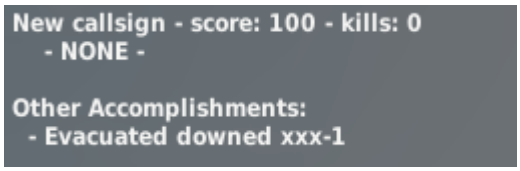
there are actually two soldiers: a pilot and an infantry. The pilot will disappear after a short while, and the infantry soldier will remain to be eventually rescued).



Now fly your helicopter to the pilot (or use the one conveniently located at the base). Since CSARManager is loaded, the procedure is the same as described there.

Once you have picked up a pilot, return to the CSARsafe zone (land near the fire truck in Senaki-Kolkhi) as soon as you touch down, the message that you have successfully evacuated a pilot appears.

Now choose communications→Other→Score / Kills to view what you have logged. Since rescuing a downed pilot counts as a feat rather than kill, you'll it listed under that category.



Note that in above screenshot we evacuated the fixed-mission pilot in Senaki-Kolkhi because I'm lazy.

8.49.2.2 ME

The main point about this mission is that there is nothing to show here. Simply by adding the two modules "autoCSAR" we got the automatically generated SCAR missions for all those hapless Eagle drivers, and CSARManger automatically submits any completed CSAR mission to PlayerScore (note that in order to view the scores, you must also add PlayerScoreGUI to your mission.

8.49.3 Discussion

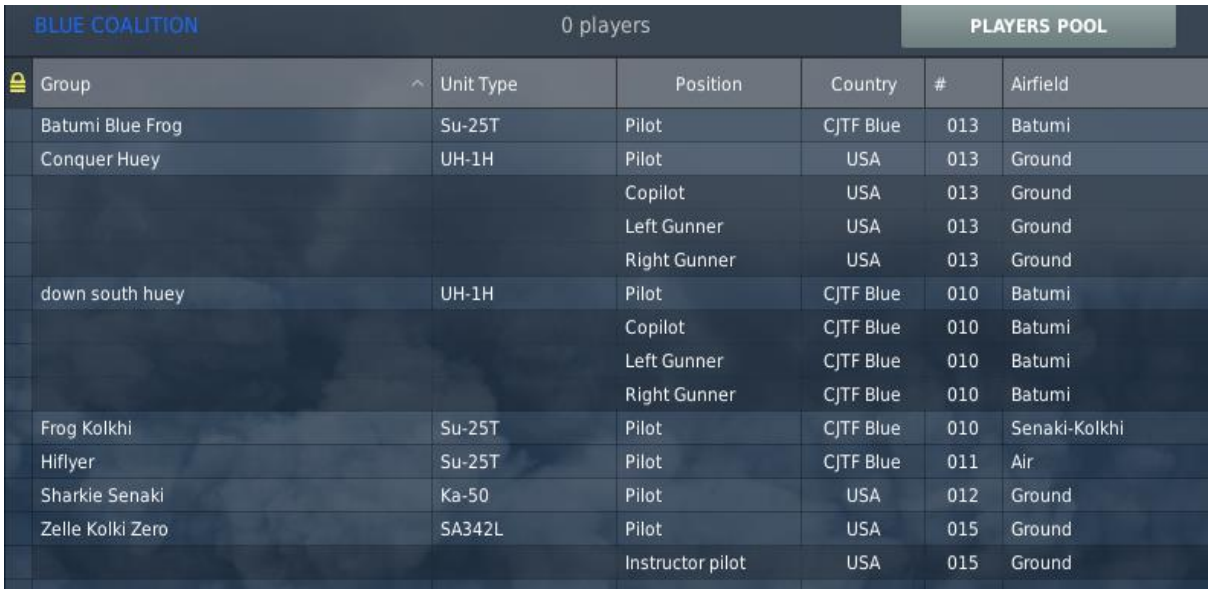
You may be wondering that, since autoCSAR and limitedAirframes have similar functionality (limitedAirframes automatically generates CSAR Missions for player (and only player) pilots), if they can work together. The answer is yes – both modules are aware of each other, and limitedAirframes automatically defers to autoCSAR for creating CSAR missions.

8.50 Slot-Blocking and You (ssbClient)

8.50.1 Demonstration Goals

This mission demonstrates how you can use intelligent slot blocking to introduce new and exciting strategic elements to your missions like aircraft that only become available after the player(s) capture airfields / FARPs

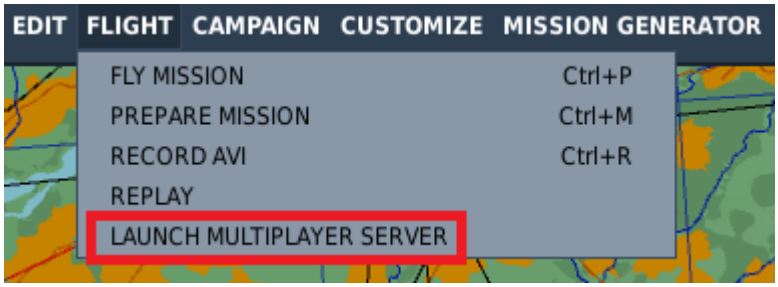
THIS MISSION REQUIRES THAT THE HOSTING SERVER HAS SSB INSTALLED



Group	Unit Type	Position	Country	#	Airfield
Batumi Blue Frog	Su-25T	Pilot	CJTF Blue	013	Batumi
Conquer Huey	UH-1H	Pilot	USA	013	Ground
		Copilot	USA	013	Ground
		Left Gunner	USA	013	Ground
		Right Gunner	USA	013	Ground
down south huey	UH-1H	Pilot	CJTF Blue	010	Batumi
		Copilot	CJTF Blue	010	Batumi
		Left Gunner	CJTF Blue	010	Batumi
		Right Gunner	CJTF Blue	010	Batumi
Frog Kolkhi	Su-25T	Pilot	CJTF Blue	010	Senaki-Kolkhi
Hiflyer	Su-25T	Pilot	CJTF Blue	011	Air
Sharkie Senaki	Ka-50	Pilot	USA	012	Ground
Zelle Kolki Zero	SA342L	Pilot	USA	015	Ground
		Instructor pilot	USA	015	Ground

8.50.2 What To Explore

8.50.2.1 In Mission



Load the mission in Mission editor, and unlike other missions, start the mission by choosing Flight→Launch Multiplayer Server. This is required because SSB, the server-side module that controls access to slots, is only active when you play missions in multiplayer. This means that any mission that uses ssbClient must be run in MP, even if it's only designed for single-player (DML automatically supports multiplayer, so this is not a limitation).

Once the slot selection dialog appears, make sure that the mission is running (if it's paused, there is a (small) chance that ssbClient did not have a chance to block all aircraft)



Choose blue's "Frog Kolkhi" – only to be rejected. If you look closely (it may be difficult to read), the server's message log on right side of the screen shows a message as response to your choice: "Sorry, <user name> Slot CURRENTLY DISABLED – pick a different Slot!

Note:
If you can enter the Frog at Kolkhi (which should be closed), there is probably something wrong with your SSB installation. Exit the mission, and make sure that you have installed the SSB script correctly in your /Hooks folder, then try again

Try “Batumi Blue Frog” next. Again – no dice: when the mission starts up, Kolkhi is closed, and Batumi is owned by the red faction, making it impossible to slot there. So, to get the ball running, choose “Hiflyer” – that aircraft is airborne, and can always be entered.

As soon as you have entered the cockpit, go to communications→Other→Airfield Control and choose “Open Senaki-Kolkhi”. This opens Senaki-Kolkhi, and allows access to all aircraft on that airfield – at least those of the owning coalition, which happens to be blue.

Now press ESC and try choosing the “Frog Kolkhi” again. You should now be able to enter that frogfoot’s cockpit.

After a short while, you’ll see a message that Batumi was captured. As soon as that happens, try to enter “Batumi Blue Frog”. Again, you will be successful – Batumi now belongs to the blue coalition and can be entered.

If you have access to (and can fly) the Huey you can now try to re-slot into the “RED Conquer Huey”. If you are quick enough (and the FARP hasn’t yet been captured), you should be able to slot into it, and fly it away. Set it down somewhere to the east, and wait for the “FARP Captured” message.



Now slot into “Conquer Huey”. It’s another Huey, albeit blue, taking of from the same FARP (freshly conquered by blue). Now try to slot into the RED Conquer Huey” again – which is denied. The FARP now belongs to blue, and no red-faction aircraft can spawn here.

8.50.2.2 ME

Default Behavior: Capture & Block



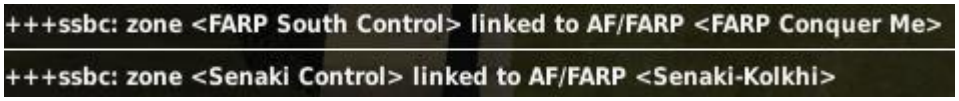
Most of ssbClient’s magic is fully automatic. As soon as an airfield or FARP is captured, this is detected, and all relevant slots are automatically opened closed. We demonstrate this with Batumi Airfield and the FARP south of Senaki-Kolkhi. Both are initially owned by Red faction and trying to slot an aircraft there is prevented by ssbClient and SSB.

FARPs and airfields are captured when there are opposing ground forces within a 2km radius, and no friendly forces to defend. In the demo, this happens after roughly 1 minute for Batumi (when a blue BTR-80 enters the 2km radius unopposed), and at the 8-minute mark for the FARP south of Senaki-Kolkhi.

Airfield/FARP Control: Open and Close

ssbClient supports ‘opening’ and closing airfields via Zones with an ‘ssbClient’ attribute. Unlike most other zones, ssbClient Zones do not need to be told which airfield they control, they automatically attach themselves to the nearest FARP or airfield. This may lead to confusion if you add FARPs close to or on the premise of airfields (which is entirely legal). In

those cases, use the zone-local ‘verbose’ function of the ssbClient zones to see which airfield or FARP the zone attaches to:



So, in order to attach a ssbClient zone to an airfield or FARP, simply move it close enough to it’s vicinity. You can then add the attributes to control the airfield. In the example to the right we show the ssbClient zone that controls Senaki Kolkhi:

Name	Value	
ssbClient		
open?	openSenaki	
close?	closeSenaki	
openOnStart	no	

open? and close? are the inputs that open and close the airport when triggered. “openOnStart” with the value false (no) is the reason why Kolkhi wasn’t available on mission start to us. If you omit this attribute, the airfield is automatically open. In order to open and close Senaki-Kolkhi and the FARP slightly south, we simply use a radioMenu to generate the required inputs (nothing special going on there).

8.50.3 Discussion

Remember that even though we can open and close an airfield, this does not mean an open airfield is open to anyone. Open Senaki-Kolkhi and try to enter “Senaki Red Frog” – a red-owned Frogfoot. Even though we opened the airfield, it still observes the ownership rule: that slot is blocked until red conquers the airfield.

So, what happens if the airfield that you slotted from changes hands? While you remain in that airframe, nothing. As soon as you leave it, however, that slot is blocked to your coalition until your side re-gains the airfield.

Challenge I

Note that this demo does *not* demonstrate ssbClient’s ‘single-use’ ability. Try to activate it, take plane from Senaki, crash it, and try to use it again. If you did everything right, that slot should be blocked after the crash. (Hint: you’ll need to add an attribute to ssbClient’s config zone **and** may need change a setting in the server’s SSB source)

Challenge II

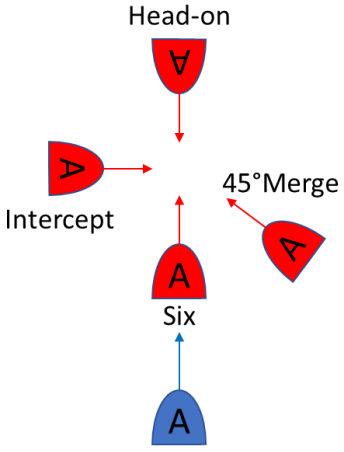
Now try to make crashed planes’ slots become available again after 3 minutes. (Hint: you need to add another attribute the config zone.)

8.51 BFM Combat Trainer (moving zone, cloner & useHeading)



8.51.1 Introduction / Demonstration Goals

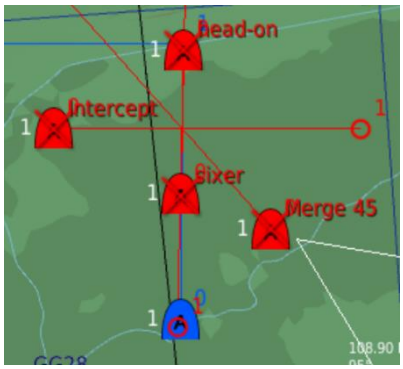
For many (especially training) missions it is desirable that the object of your training (a.k.a. "Target") is positioned at a certain location and bearing. For example, to train how to engage an enemy that comes straight at you, that enemy must first be placed directly in front of you (meaning: in the direction you are flying), and it must be heading in your direction. A traditional DCS solution would be to tell the pilot to go to a certain location, come to precise heading, and then activate the target when all criteria are met.



This is not the DML way – let's make it far more convenient for us and the player. So, let's start small: we want to create a BFM (basic fighter maneuver) trainer that allows the player to spawn planes in four different positions:

- We want to spawn at their six – directly behind them
- We want them to spawn in front of us, heading in our direction
- They should spawn crossing from our left to right (right angle intercept)
- We want to fly a 45-degree merge with the enemy coming from our right

Setting this up in ME isn't difficult at first:



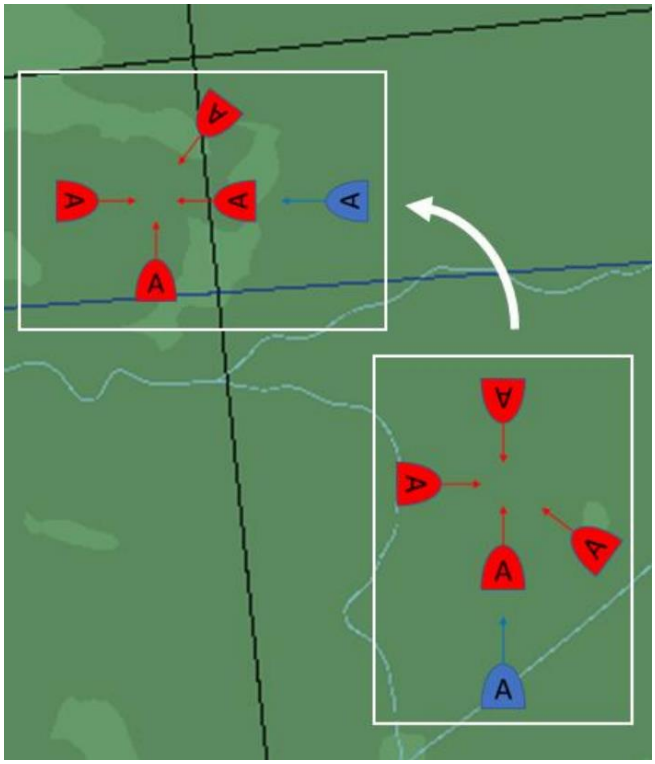
Simply place the enemy planes relative to the player's plane, set their routes to meet the requirement, use a cloner for each and activate each group with a trigger (driven conveniently with a radioMenu) – and we should be set, right?

Well, not so much. We realize the problem with this setup as soon as we enter the cockpit. There are several rather big problems:

- All positions are fixed. For this to work, the pilot has to be at the initial point or the planes spawn in wrong locations relative to the player.

- All courses/routes are fixed. If, for example, the player heads due east, the Head-on plane now won't spawn in front of us but at a right angle to our left, with a course set to pass behind us.

That is why traditionally, missions use the 'Initial Point' method where players need to fly to a pre-arranged point and come to a precise heading for the mission to line up – precisely what we have set out not to do.



So, how are we going to tackle this problem? What we are looking for is a solution that moves the spawners together with the player unit, and keeps them in place relative to the player's orientation. It rotates the 'entire picture', so to speak, including the plane's headings and routes with the player plane

That's what we want to learn how to do here. So, hop into the Frogfoot and explore what the mission can do for you

8.51.2 What To Explore

8.51.2.1 In Mission

Enter the cockpit of the Frogfoot. You are heading straight North at an altitude of 2000m. Remember the mission layout from the introduction, you remember that the 'Head On' plane (and cloner) were positioned 4km (2.5 miles) to the north of the blue player plane, and that plane's route was set up to head straight South.

Turn to the left until you are heading out to sea (straight west 270 is perfect). Try to remain at 2000m altitude (remember the Frogfoot uses the metric system). Now choose communications→Other→Spawn Enemy→Spawn Head-On.

A red A-10 poofs into existence, coming straight at you! Check its label to confirm that this is indeed "Head-On One", the unit that in ME was positioned north of you and was heading your way. So, it appears that this mission succeeds in the following:

- Moving the cloners with the player unit
- Keep the cloners in position relative to their original bearing from the player
- Rotate the spawned planes to keep their relative heading to the player plane
- Move and rotate the spawned units' route accordingly

Let's try the 'Six' spawn. Continue westward until you are close to the coast, make your altitude 2000m and then choose Communications→Other→Spawn Enemy→Spawn on their Six. Some 0.8 nm in front of you and heading the same direction as you spawns an A-10A named Sixer One. You have spawned directly on its six, so close the distance and try some formation flying behind the enemy, or try to down them.

You can now also try the other (more difficult) intercepts: at right angle (the hog will come from your left), and 45 degrees (the Hog comes from your right). Experiment with your altitude (the Hogs all spawn at 2000m / 6560ft) so you can use that to your advantage.

Note that – since we use standard DML cloners to spawn the Hogs – you can spawn the same Hog multiple times. Their skill is set low, and all they have are their guns, you should have no difficulties in besting them should you decide to fight (your Frogfoot is equipped with multiple missiles, and two gun pods in addition to its cannon).

8.51.2.2 ME
We achieve most of the magic by a feature that we get for free from DML general Zone Abilities: moving zones. Since this is a core ability, it's available to any module that uses zones – and (almost) all DML modules use zones.

Let's recall that any zone with a linkedUnit attribute turns into a moving zone that follows the unit named as value in the 'linkedUnit' attribute around.

DML offers some additional abilities to moving zones that we use here. By default, a moving zone centers on the unit that it is following. This essentially mimics DCS's moving zones, and can be very useful.

Additionally, DML offers the 'useOffset' attribute, which makes a zone follow the master unit around by the same offset. This would keep the cloners always at the same position relative to the player unit, but it would not account for any changes in heading.

Finally, the 'useHeading' attribute is what we are using here: it always keeps a zone at the same bearing from the master unit, making it orbit around the master unit as that unit turns – and this is what we are using in this mission.



Zone initially set at 4 o'clock position rotates with master unit to stay at 4 o'clock

So, the linkedUnit/useHeading combo accounts for the zones always staying in the correct distance and bearing relative to the master unit (named 'anchor' in this mission). The final piece of the puzzle is the 'moveRoute' attribute that we activate for the cloner. When a cloner clones a group, it also clones the group's route. By default (without adding the 'moveRoute' attribute, a cloner modifies the route's initial waypoint, leaving all others unchanged. This is so that no matter where you clone a group to, they rally at the first waypoint and then move along the rest of the same route.

Name	Value
linkedUnit	anchor
useHeading	yes
cloner	
clone?	doInter
moveRoute	yes

The 'moveRoute' changes that – it usually moves the entire route by the offset. However, if cloneZone detects that its zone is linked to a master unit and also has useHeading set to true (meaning that the cloner is set to rotate around the master unit), it also rotates the entire route in synch with the zone (if the zone rotates 50° around the master unit, the group's route is rotated by 50° as well). This is what we use here: That way, all routes and groups rotate with the cloners and remain aligned with the master unit

8.51.3 Discussion

Moving Zones minutiae
We are using zones that are linked to a player unit. When you revisit this document's section on moving zones (it's part of 'Core Abilities'), you'll notice the following comment when describing the linkedUnit attribute:

That unit must exist at the beginning of the mission, or the linked zone remains at its ME location until the unit exists and becomes linked (at which point it moves to the unit's location)

All zones in this mission link to the unit named 'anchor' which is the player Su-25T. And here's the rub: client planes do not exist at the beginning of the mission, and the initial link to the player unit fails. It's established only once you enter the cockpit, which is usually a few seconds into the mission. To see how DML handles this, click on the Trigger Zone 'Intercept' (the one on the left of the group crossing to the right), and add a verbose = true attribute, then start the mission. Pause at the aircraft selection screen and inspect the text that appeared on the right

A few additional lines have appeared on the screen, with diagnostics for the Intercept zone (as expected, as it has verbosity enabled). One line is of particular interest to us:



This warns us that the "Intercept" zone attempted to link up with unit "anchor", but did not find any unit with that name in the game. DML handles this gracefully, and will try continuously (once per second) to establish that link. The important part to remember here is that as long as moving zones are unlinked, they remain where they

Now enter the cockpit, and sure enough, a new message appears:



This tells you that DML now successfully links Intercept to anchor, and since it has enabled useOffset or useHeading, the zone also establishes the offset and distance.

Frogs and Hogs
We used Frogs and Hogs in this mission for simple reason:

- The Frogfoot is the only jet that everyone is guaranteed access to, so I know you can (theoretically) fly it to try out the mission
- The A-10 is sufficiently benign (and slow) to ensure that you survive at least your initial encounter.

This allows you to inspect the anatomy of a simple, effective BFM trainer. Of course, your next step is to kick up the difficulty.

Challenges

Kick up the difficulty:
Change the player plane to your favorite module, and then change the red planes to some foes that are appropriate to your level of flight expertise. Also, kick up their skill and give them some real teeth. Then see if you can survive the encounters.

Pain in the Neck
Set up a trainer that spawns a competent, armed enemy plane at your six on command. Then survive.

Ground Pounder
Set up a mission that, on command, spawns random ground targets 2 miles straight in front of you. The spawner should be able to choose randomly from three different groups of enemies. If you like it tough, add some AAA to the mix.

8.52 Formation Trainer (messenger, counter, moving zone)

8.52.1 Demonstration Goals



The pieces that were needed to fall in place to make this little demo happen were surprising to me – I wanted to write this particular kind of mission for some time, and now that DML has all the required abilities in place, I'm really happy to see how elegant and easy building it has become in DML.

This deceptively simple demo demonstrates a couple of DML key abilities that are impossible to achieve ME-only, would require some heavy Lua with most other frameworks, and can be done with just a couple of zones in DML

All this mission is to accomplish is this:

- Mark a moving area close to a lead plane where the player has to fly to, and remain there
- When not inside the area, provide constantly updating directions
- When inside, count the time, and provide constantly updating feedback to help the pilot to stay inside
- At the end, provide the total number of seconds inside the area

When implemented, one module brings 90% of the lifting power ("Messenger"), while UnitZone and DML's moving zone carry the rest. "Changer" makes a strong appearance, although only for cosmetic purposes, as "Counter".

8.52.2 What To Explore

Note:
This demo requires above-average aircraft control for precision-flying. If you have never done formation flying yourself, the experience will be humiliating, spectacular and not much fun. Don't try to force yourself to fly the Frogfoot that well. There are two proven ways to make the mission easier on you:

- **Enlarge the Trigger Zone 'fZ'** to 300 feet (from the mission's 30 feet) to relax the constraints and help you better avoid crashed

- **Change the aircraft type** of all planes to a type that you are more comfortable with. All you need to do ich open the mission in ME and change the aircraft types from groups “Follow” and “Formation”. Warning: make sure that you change the types of both groups to the same aircraft type, as flying formation in dissimilar aircraft types can sharply increase difficulty.

8.52.2.1 In Mission

Enter the Frogfoot (or whatever plane you preferred). When the mission begins, you hear a brief voice message from Lead, instructing you to form up on this starboard (right) wing.

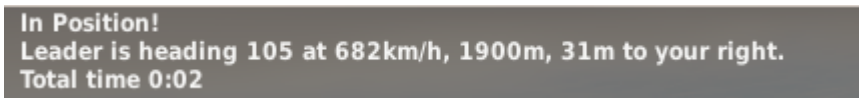
More importantly, a constantly updating block of text appears in the upper right corner of your screen similar to this:



That text box is this mission’s crown jewel, as it shows off most of messenger’s advanced unit-relative capabilities. For the time being, though, simply note the following

- The text updates once a second
- It shows Lead’s current heading, speed and altitude (note that depending on the settings of messenger’s ‘imperialUnits’, you may see different numbers and units. The image above shows some backwater country units, you can also use SI for better clarity)
- It shows the distance and direction to your pane’s target spot behind Lead
- It shows how quickly you are closing in on that position. A negative value means that you are moving away (i.e. Lead is faster than you are, or you are heading in the wrong direction)
- It shows Lead’s aspect – this serves no other purpose than showing off what messenger can do
- It shows a total of the time you have spent inside the target area. If you have increased Trigger Zone “fZ”’s radius, getting inside and staying inside will be much easier.

Now comes the difficult part (if you have never done formation flying before): approach Lead until you are close enough that the text block changes to “In Position!”



(Obviously, I was lazy when I took the above screen shot and made the target area much bigger than in the mission). We now see that messenger has switched to displaying different information:

- Lead’s heading, speed and altitude (here in SI units as they should be)
- Distance to the exact location of the assigned position
- Direction to your intended position (ahead, left, right, behind), as seen from your current heading
- Total time inside the zone in Minutes:Seconds

Should you get too far away from your designated location, the display switches back to the 'approach' display; when you gat back into the zone, back comes the 'In Position' display, continuing the total time in zone.

Once you get to Batumi, you are instructed to land. The approach/track display fades, and once you land you get a report of your total time on your position.

8.52.2.2 ME

The mechanics of this mission are a lot simpler than you may fear: it's pretty much determining if the player plane is inside or outside of the Trigger Zone "fZ", and two slightly different messengers (one for "inside" and one for "outside" the "fZ"). Most of the magic is delivered by messenger's Wildcard abilities that can access and calculate directions, bearings etc.

Let's take the mission apart:

"The Spot"

This is quite simple – it's small Trigger Zone ("fZ") that is linked to Leader's plane (*linkedUnit*), so it always moves with it. It also uses the *useHeading* attribute to it stays in the same position relative to Leader's heading, so when Leader changes heading, zone "fZ" turns with Leader.

Name	Value	
linkedUnit	Leader	
useHeading	yes	



Zone initially set at 4 o'clock position rotates with master unit to stay at 4 o'clock

Inside/Outside flags

To determine if the player's aircraft is inside or outside of the trigger zone 'fZ' we use a *unitZone* that only triggers on player's plane 'followOne' (*lookFor* = players, *matching* = "Follow One").

unitZone		
lookFor	Follow One	
matching	players	
uzDirect	inZone	
uzDirectInv	outZone	

We also use the direct and inverted direct outputs *uzDirect* and *uzDirectInv* to simplify controlling our messengers (see later). All we need to know here is the output *uzDirect* (which writes to flag inZone) is set to 1 as long as the player plane named "Follow One" is inside 'fZ' and is set to 0 otherwise. The inverted output *uzDirectInv* (which writes to flag outZone) has opposite values: it's set to 0 when the plane is inside, and 1 when outside. We'll use that later to control which messenger is active

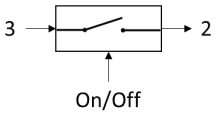
The upshot here is that flag *inZone* is 1 whenever the player is in position, and 0 otherwise, while the flag *outZone* is 1 whenever the player is not in position and goes to 0 when they move into position.

Or put differently: whenever inZone is 1, outZone is 0. Whenever inZone if 0, outZone is 1. *inZone and outZone always have opposing values.*

Open when the other is closed: two Gates

Centerpiece of this mission are two messengers: one for when the player is outside ‘fZ’, and one for when they are inside. Only one of them must ever trigger at the same time, so we use a “Change” module in Gating configuration to control which one will activate.

Gates are controlled by the *on/off* input: whenever the value on that input is set to 1, the gate is open, and the information from input can flow to output. If the value on then *on/off* input is 0, however, the gate is closed, and the output remains unchanged, no matter what happens at the input.



We use that to our advantage with the two messengers. The steady heartbeat pulse that arrives every second would, when connected directly to the two messengers, cause both messengers to fire at one, and one of them would overwrite the other. So we need to find a way to make sure that at any given time, at most one messenger receives the heartbeat, while the other messenger’s input must be blocked. And that is where the gate comes in: we can use it to block the input to those messengers.

Now recall that above we set up the two flags inZone and outZone in such a way that they always have opposing values: 0 when the other is 1, and vice versa. And this we now use to control the two messengers: the one that should only display text when the player is inside the zone has the “inZone” flag (which is 1 when the player is inside the zone, 0 otherwise) feed into that zone’s gate *on/off* input, while the messenger that should only trigger when the player is outside, wires the flag “outZone” (which is 0 when the player is in the zone, and 1 when outside) to it’s on/off input. Since we know that the two flags have opposing values, we know that the gates are always in opposing states: when one is open, the other is closed.

Name	Value
change?	heart
out!	*beat
on/off?	inZone

Name	Value
change?	heart
out!	*beat
on/off?	outZone

The result is that while a gate is open, the heart flag (which is driven by a pulser) flows through to local *beat, which is then wired into the messenger’s trigger input.

The Messengers

Which brings us to this missions workhorses: the two messengers of which we know only one will ever trigger at the same time (which is ensured by the gates, above). Both messengers are triggered whenever their local *beat flag changes, and we’ve seen before that this only can happen when the gate is open.

messenger?	*beat
responses	in front, to the right, behind,
message	Leader is heading <hdg:Lea
unit	Follow One
clearScreen	yes
imperialUnits	no
messageOff?	landBatumi
timeFormat	<m>:<:s>

So let us begin by analyzing the messenger that triggers whenever the player is outside of the zone:

We see that it's set up to only send messages to the unit Follow One (the player's plane) via the *unit* attribute. The fact that this messenger knows the unit it is communicating to is important as we'll later see.

We note that we can turn the messenger off entirely (even if inputs are passing the gate) with a signal on the *messageOff?* input which is fed into by flag landBatumi (see later). So note that we have wired in an ability to turn the messengers off.

Each time that the messenger is triggered, it clears the screen (*clearScreen* = true), which gives it the self-updating appearance. And makes it impossible for the screen to share any other information – even if we wanted to: the screen is cleared every second.

We change *timeFormat* from its default to interpret the value given as full minutes (meaning they won't reset at 60) and standard seconds (reset at 60).

And we see that *imperialUnits* is turned off, which will cause messenger to use real (SI) units like meters when calculating speed or distances.

Before we delve into the message itself, we note that we also have given four different possible *responses* as follows:

```
in front, to the right, behind, to the left
```

Message is set up as follows:

```
Leader is heading <hdg:Leader> at <vel:Leader>km/h, <alt:
Leader>m.<n>Your assigned position is <rng:fZ> <rbea: fZ> of
you, closing with <pcls:Leader>m/s. Lead aspect is <asp:
Leader><n>Total Time in Position: <t: secs>
```

In other words, wildcard bonanza. This is a possible result,



and now let's go through them one after the other to see what they do:

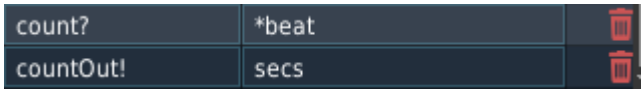
- `<hdg:Leader>` looks up the unit named "Leader", calculates the **heading** in degrees and returns it. 106 in the example.
- `<vel:Leader>` looks up unit "Leader", and calculates it's **velocity**. Since *imperialUnits* is false for this zone, the result is returned in km/h. Unlike the example, which uses backwater units
- `<alt: Leader>` shows that you also can insert blanks between the keyword alt: and object "Leader" – and of course will make messenger look up unit "Leader" to get its barometric **altitude**. Since *imperialUnits* is false, the value returned is in m (unlike in the example above)
- `<rng:fz>` returns the **range** (in meters because *imperialUnits* is false) to the center of zone 'fZ'
- `<rbea:fZ>` uses messenger's unique '**winding response**' ability. The responses as given above (in front, to the right, behind, to the left) are wound around the compass rose and indexed **by** the **bearing** to the center of zone "fZ". In the example that is 'in front'.
- `<pcls: Leader>` looks up unit "Leader" and calculates its velocity. Then, messenger looks up the player's unit to whom this message is sent (supplied with the

unit = “Follow One” attribute) and calculates that plane’s velocity. From both, messenger then derives the **closing velocity**. Since the keyword is ‘pcls’, the **precision** form is used. Since *imperialUnits* is set false, the result is given in m/s, exact to one decimal.

- `<asp: Leader>` looks up unit “Leader”, then the messenger’s target unit (player), and determines where Leader’s nose is pointing to (the ‘**aspect**’ of unit Leader with regard to player’s unit).
- `<t: secs>` reads the value of flag “secs” and uses timeFormat to convert it to a **time value**. Since we gave a time value for this messenger, the value of flags will be converted to minutes:seconds, with minutes not resetting at 60, meaning if the flag “secs” held the value 4346, the formatted time would display “72:26”

When we inspect the inZone messenger, we find that it’s essentially the same, except that it omits some wildcards, and uses slightly different responses for the <rbea: x> wildcard.

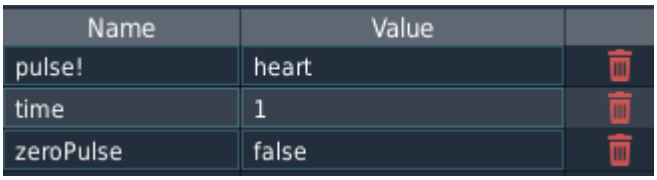
Counter:



There is one addition to this zone, that outZone didn’t have: this zone also stacks a counter module, that’s keyed to the same input flag as messenger. So, whenever this messenger (inZone) is triggered, the counter is also triggered, and increases the flag “secs” – the value that is displayed in the message as a time value.

Since we know that the heartbeat pulses once a second (or see below), we have an easy method to calculate the number of seconds the player is inside ‘fZ’: each time that the messenger triggers, we increase the value of ‘secs’. This increase to ‘secs’ only happens when the player is inside zone ‘fZ’, but not when outside.

Heartbeat



The final piece to the puzzle is simply a pulser that gives a signal on the flag “heart” once a second. It’s used to drive the per-second update of messenger. The pulser itself is nothing special: time of 1 makes it increase the flag “heart” every second, and we skip the very first pulse for purely cosmetic reasons (so messenger and the other modules have some time to sort themselves out).

Turning off the messengers entirely

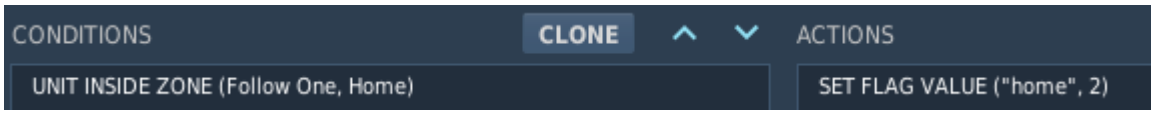
This is some more polish for the mission: we want to be able to display the total time that the player spent inside the target zone. For this, however, we need the messengers to stop triggering because their *clearScreen* = true flag would immediately erase whatever we display and replace it with the approach/inside statistics.

So we set up a standard ME trigger zone, and as soon as the first blue plane enters it, we invoke an action to set the flag “landBatumi” to 2



This flag is wired into both messengers messageOff? inputs, and a change on that flag will cause both messengers to shut up.

So the last messenger's message will time out while the player has time to set up for final and land at Batumi, and as soon as they are inside the 'home' trigger zone, another standard ME trigger sets flag home, which in turn triggers the 'home' messenger



This messenger is quite standard, and merely sums up the time inside 'fZ' by displaying the value of flag "secs" formatted by timeFormat (the same format as the two live messenger used). The message is kept on-screen for 6 minutes (360 seconds) for good measure so that the player can carry out a missed approach, land, taxi to parking, and still see his final in-zone time.



8.52.3 Discussion

Why the spoken dialogs?
When you fly the mission, you're sure to notice my dulcet voice treating you to a kraut rendition of commands. Why did I add these? Polish, of course. But it's also necessary because the way that the messengers are set up, there is no way to place another message on the screen: it would be immediately erased the next time that the heartbeat pulse arrives (at max 1 second later). Keep this in mind when you use a regularly updating message: your screen estate for messaging is limited. That is also why we stop the two messengers when we reach Batumi and from that point on are able to use the screen for normal messages.

Challenges:

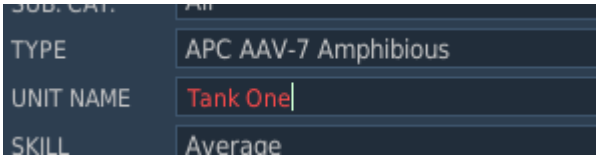
Imperial Units
The world is a harsh place, and sometimes you need to do things that are plain painful. This challenge is one of them: Change the approach/inside messengers to correctly give all information in ~~inferior~~ imperial Units.

Other Plane Types
This one should be quick, and merely a demonstration of how incredibly versatile DML is: change this Trainer to a different aircraft type. Let the messenger also mention the plane type of the Leader

8.53 Hope you guess my name (Cloner Name Schemes)

Note: This topic is for medium to advanced mission designs.

Unit names in DCS missions have a strange property. Most mission designers are somewhat aware of this, and it rarely becomes so relevant that it rises to the level that they have to actively manage it: each unit's name must be unique. This means that at any time during a mission, there is at maximum one unit with a specific name.



If you try to assign a name to a unit that is already given to another name, Mission Editor won't let you (name turns red) until you change the name to something that does not conflict with any other unit's name. The reasons for this are never fully explained, and I assume that this is for historical reasons (it may relate to the fact that the *other* unique identifier, unit ID, was never opened to Mission Editor, and thus left DCS with two independent unique identifiers for the same unit).

Since ME enforces unique unit names, this subject isn't relevant for anyone who uses 'vanilla' ME. Once you spawn units dynamically, however, this subject becomes relevant.

So what happens should you spawn a unit with a name that another existing unit already has? The answer is simple: to resolve possible name conflicts, if you spawn a *new* unit named "x", an *existing* unit named "x" is immediately removed. Since 'vanilla' missions can't spawn units, that's not an issue there. When your mission does spawn units though, the onus is on the mission designer to ensure that no name conflicts arise.

DML's spawn modules all provide safe naming schemes by default, so in the context of 'vanilla DML', you would never run into this issue.

And this is where it gets interesting. There are cases where you want to control the name of units that your mission spawns - for various (advanced) reasons.



8.53.1 Demonstration Goals

In this demo we look at cloners that use the optional '*nameScheme*' or '*identical*' attributes to achieve some advanced features that are inaccessible to mission designers otherwise:



- *Allow ME conditions (rules ) to trigger on cloned/dynamic units*
  This is the most common and important use case: In order to use a standard ME Trigger Condition (rule) in a ME trigger, you need to know the unit's name when you edit the mission. This usually precludes being able to use trigger conditions with cloned (dynamic) units. Unless, of course, you are able to spawn clones with that name.

- *To replace existing units with "fresh" ones*
  Sometimes, usually for long-running (server) missions, you may want to periodically 'refresh' units to simulate repairs/reinforcements/resupply. Although cloners supply other means for this, using same-named units that replace their previous incarnation is a simple way to do so.

- T*o make "deserter" units (units that switch sides) while the mission runs*
  When you place units in ME, you always assign a coalition to that unit. While the mission runs, those units can't switch sides. With DML cloners, we can do that by spawning the same unit with the same name, but belonging to the other faction. The newly spawned unit then replaces the existing unit that belonged to the other faction.

- *To 'teleport' units across the map*
  We can take advantage of the fact that an existing unit is removed when a unit of the same name spawns someplace else

- *To randomize group compositions*
  We can make groups with different compositions that spawn in-game all with the same name (important if they need to work with vanilla ME trigger zones or conditions)

8.53.2 What To Explore

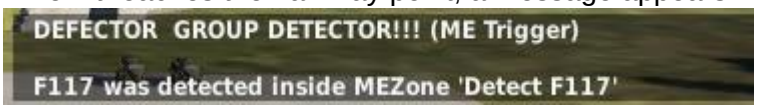
8.53.2.1 In Mission

So let's start with a rare, but cool effect that you can't create unless you resort to DML or scripting: Have a unit 'defect', and have that defected unit still trigger standard ME-based flag conditions.



- Start the mission and use F2 until you are observing the F-117 Nighthawk.
- Note that it's a member of the blue coalition.
- The Nighthawk is undergoing a cold-start.
- Accelerate time, and around the 1:40 time mark, int starts moving to make its way to the runway.
- *Immediately* go back to normal time,
- continue your observation. The Nighthawk slowly moves forward towards the taxiway straight in front of it.

- When it reaches the half-way point, a message appears:



  This message is triggered by a 'vanilla' ME trigger condition; simply take note that the blue unit named "F-117 Parked" triggered this condition in ME.

Now re-start the mission (the following will only work correctly if you restart the mission) and

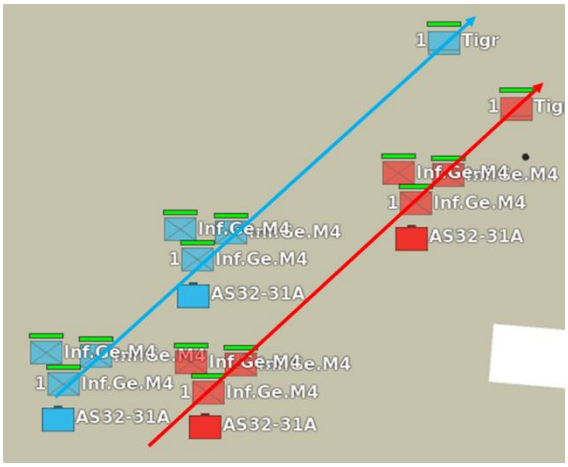


- Go to communications→other→F117 (Defector) Test and choose "Spawn F117 as RED"
- Use F2 until you are observing the F117
- Note that the Nighthawk now is a member of the RED faction, yet it has the exact same name ("F117 Parked") as the blue one it replaced.
- Like before it's undergoing start-up, so feel free to accelerate time until it starts moving towards the taxiway
- Take note that as before, when the Nighthawk reaches about halfway, the same message as before appears. Evidently, the red unit triggers the same condition.

Now re-start the mission (yet again) and



- Switch to the F10 map view
- Pan down (south) and look for the three "Tigr" vehicles that are ambling across the runway towards the tarmac where your Su-25T is parked.
- Notice that each is of a different faction: One is Red, one blue, and the third red.
- While on map view go to communication→other→Tigr Spawns. Note that there are three possible choices: spawn from blue, spawn from red, and spawn replacer.



- First, alternatingly, choose 'spawn from red' and 'spawn from blue' a couple of times (at least twice each).
- Note that each time you choose 'spawn from …", a group of infantry and a static object spawns
- Note that each time a group and an object spawns, they belong to the same coalition as the spawner
- Note that the location where the group of infantry spawns moves with the Tigr.
- When you click on the spawned infantry (not the static object) units to display their names ('callsign' in DCS lingo), notice that each unit's name is of the pattern "b-spawn-x" for the blue units, and "r-spawn-y" for the red units, with x and y each starting at 1 and growing.

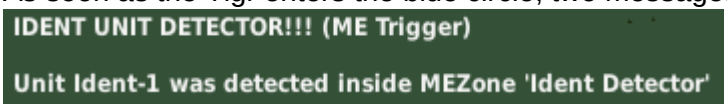
- Also note that, for reasons unknown, the static objects display no name, just their type (AS32-31A)
- Now try the same with the ‘Spawn Replacements’ option. Choose this option a couple of times (say four times).
- Note that, just like before, a group (now neutral) of infantry and a static object spawns
- Note also that the spawn point moves with the neutral Tigr
- Note especially that as soon as you sawn a new group, any previously spawned group is removed (this holds true for both units and static objects)



- When you inspect any of the spawned units you will find that their name (“callsign”) is always the same “replacer-1” through “replacer-3”. This holds true even if you re-spawn again with ‘spawn replacer’.
- Looking very closely, you’ll also notice that unlike the clones spawned by the red and blue cloners, for some reason there are only two infantry instead of three.

Now let’s turn our attention to the last two special effects:

- Pan the F10 map view until you can see the large blue circle on runway 09 in Senaki
- Choose communication→Other→Identical Spawns.
- Choose UNIT IN ZONE Test
- Notice that a new Tigr appears that starts moving towards the blue circle
- As soon as the Tigr enters the blue circle, two messages appear:

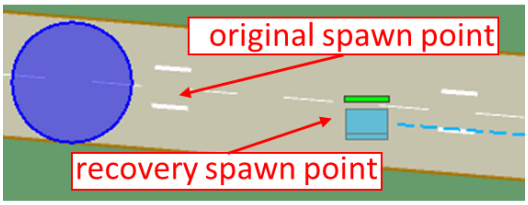


and



And another neat trick:

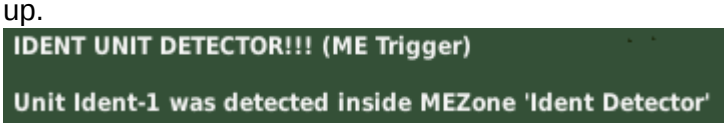
- Make sure you can see the blue circle in F10 map view on runway 09 in Senaki
- Choose communications→Other→Identical Spawns
- Now choose “Spawn Recoverer”
- A blue Tigr appears east of the blue circle and starts moving east down the runway. Mark the spot where this Tigr first appeared.
- Wait until the Tigr has moved a nice distance from the point is spawned, then choose “Destroy Recoverer” from the same menu.



- Unsurprisingly, the Tigr disappears.
- Now, choose “Spawn Recoverer” again.
- Note that the Tigr re-spawns at the location it disappeared, not the location it originally spawned.

The final test is perhaps somewhat strange. Restart the mission (required) and

- Pan to the now familiar blue circle on runway 09
- Choose communications→other→Fake Name Test and choose “Spawn with faked name”.
- In front of the blue circle, a unit “Ident-1” spawns. However, although the name matches the previous “Ident-1”, instead of a Tigr we clearly see that this unit is a gas truck – definitely not a LUV Tigr.
- The spawned unit proceeds to move into the blue circle
- A message that the unit Ident-1 has triggered the IDENT UNIT DETECTOR comes up.



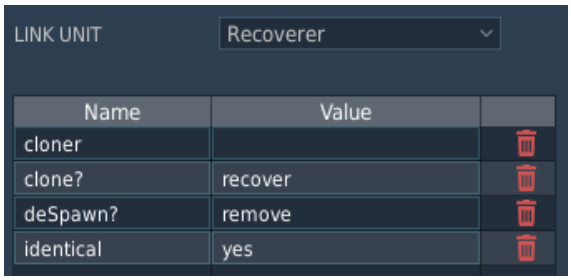
- **More importantly**, note that the IDENT GROUP DETECTOR is **not** triggered.

8.53.2.2 ME

So let’s go through the various items in reverse order

8.53.2.2.1 The position-recovering Tigr spawn

What we are doing here is taking advantage of two things:

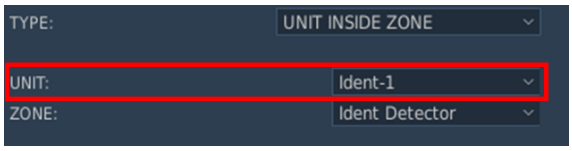


- First, we move the Tigr’s cloner (spawner) with the Tigr unit itself by linking it to the Tigr on the map. Note that we are using the ME’s LINK UNIT ability rather than DML’s *linkedUnit* attribute. By doing so, we are linking the unit to an **existing** unit in ME. This usually is problematic, since a clone zone always deletes all units that are part of its template, and that is why the Tigr is gone when after the mission starts
- Although the Tigr is removed on mission startup, its information is retained in the cloner’s template. Enter the ‘*identical=true*’ attribute. If the cloner spawns with that attribute set, it creates an exact clone of the template, including all names and ID. So, as soon as the cloner runs a clone cycle, the zone finds the linked unit, attaches itself to it, and moves with it
- When we despawn the Tigr, the zone becomes unlinked, and remains where it is.
- When the next *clone?* signal arrives, the cloner again spawns an identical clone. However, since the clone zone’s position has moved with the Tigr since the last time, the new clone is spawned at the location where it last spawned
- Because the clone is an identical clone to the one that the clone zone is linked to, it again re-attaches itself to the new clone, and starts following it around.

Note that this effect has a limitation: it only works with groups that have one-waypoint routes, as DCS currently orders all spawning groups to move WP 1.

8.53.2.2.2 Clones that can trigger ME conditions

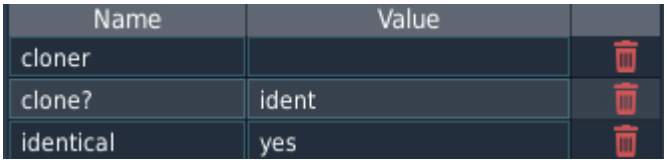
At first blush, there was nothing spectacular about this: to a unit or group triggers an ME condition. So, what, boo hoo!



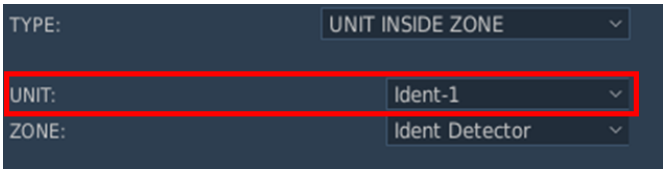
Well, if you try that yourself, you'll quickly find out that the issue is with ME's trigger condition editor. When we build the trigger for this and then try to assign the trigger condition we find that it restricts triggering units to units that already exist in the mission (i.e., that are placed somewhere on the map), and when you choose one, it is referred to by name (here "Ident-1").

This poses two problems with dynamically spawned units: they do not exist when we edit the mission in ME (else they would not be dynamically spawned), and due to the nature of dynamically spawned units, they are guaranteed to use names that are different from anything that already exists to prevent name collisions.

This is where the '*identical'* and 'nameScheme' come in. CloneZones know about the particular way that DCS works behind the scenes and make sure that clones that are spawned from zones with these attributes are made compatible with ME's peculiarities.

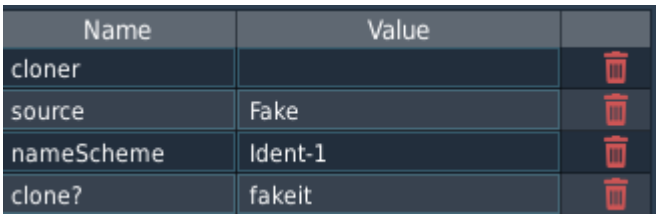


Let's start with the simpler case: when we use '*identical'*. As explained in Clone Zone's section of this document, clones that are created from a clone zone with the *identical=true* attribute use identical names and IDs for both group and units. Although in the official documentation a group's or unit's ID are only mentioned in passing, DCS-internally the IDs (and *not* the name) are critical for trigger conditions to recognize units or groups.



If you use the ***identical=true*** attribute, the cloner not only uses the exact names from the template, but also the same IDs for both units and groups. The clone is identical in every way to the originally placed group in ME, and therefore passes **ME's trigger conditions for both group *and* units**.

This is evident by the Tigr triggering ME's IDENT GROUP DETECTOR and INDENT UNIT DETECTOR conditions.



Now let's look at the "Fake Name Test" example. The spawned unit is a gas truck, not a LUV Tigr, yet it has the same name "Ident-1" as the unit we spawned in the '*identical'* example. Obviously, it's not an identical template spawn, but the *nameScheme* that we used for this clone ("Ident-1") results in a name that *intentionally* conflicts with a unit's name that we reference in ME's trigger condition.

When cloning units, the cloner automatically recognizes this, and ensures that such a unit receives the correct (Mission Editor-assigned) ID to be able to trigger ME conditions. Since nameScheme only extends to unit names, this affects only units, not groups. That is the

reason why in the demo, the UNIT DETECTOR was correctly triggered, but not the GROUP detector.

Please remember:
the name scheme used in this example is unsafe to use in normal missions. We *intentionally* use a name-conflict-generating scheme to achieve the desired effect. Be sure to re-read CloneZone's documentation on *nameScheme* before trying this stunt on your own)

8.53.2.2.3 When to use: identical vs nameScheme

The (mutually exclusive) attributes *nameScheme* and *identical* can be used to achieve similar goals: allow clones to trigger ME trigger conditions.,The way that they function are subtly different. *Identical* is easier to use (as it exactly replicates a template), and its results are somewhat more limited. On the other hand, *nameScheme* is more flexible, and can introduce strange unwanted mission results when used incorrectly.

So what are the pros and cons? When should I use which?

- *Use* identical when
  you want a cloned **group** also to be able to trigger a ME condition. The *identical=true* attribute is the only one that can achieve group compatibility.

  Disadvantages: *identical* always reproduces the entire template.

  Advantage: it works with ME trigger conditions for **Units *and* Groups**, and guarantees name and ID match-up for all units to the template.

- *Use* nameScheme *when*
  you want any cloned unit to be able to **match to a ME UNIT condition**. What CloneZone does is simply to compare a unit's assembled name (according to the *nameScheme*) to any unit name the ME knows and might trigger against. If it finds a known ME unit name, the cloner simply ensures that this unit also receives the ME-assigned unit ID so it can trigger ME conditions. This means that, unlike when using *identical*, a unitID match-up is incidental with *nameScheme* rather than guaranteed.

  Disadvantages: You must ensure that the *nameScheme* matches an existing unit's name, and that the *nameScheme* does not create unwanted name conflicts in your mission. Units cloned with *nameSchme* will not work with ME trigger conditions that work on groups (e.g. PART OF GROUP IN ZONE)

  Advantages: *nameScheme* only requires knowledge of name to replace; it can use different template to replace an existing unit simply by matching up the name of the units to replace.

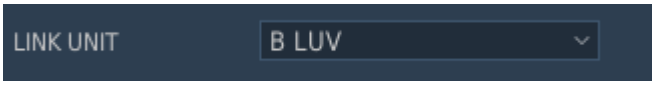
8.53.2.2.4 The Moving Spawners with nameScheme

The blue and red cloners that are attached to the Tigs are straightforward. They merely show how we can use masterOwner * to force a spawned template's coalition, and how to use a nice, easy-to-debug name scheme for units.

So lets walk through the blue cloner's configuration.

Name	Value
cloner	
source	Blue Men
nameScheme	<z>-<lcl>
owner	blue
masterOwner	*
clone?	luvB
useOffset	yes

- *source* for this cloner is a trigger zone called “Blue Men”, so everything that this cloner clones is imported from that cloner's template (incidentally: 3 infantry with M4 and a static object, the original template itself is aligned with blue)
- *nameScheme* tells this cloner that all units names, when spawned, should be constructed from the spawning zone’s name (<z>), a hyphen “-“ and a zone-local count <lcl>. As we have seen, for this cloner it results in units being named “b-spawn-1”, “b-spawn-2”, etc. Including the spawning zone’s name makes units easy to debug (you can trace the spawner where the units originated), and adding <lcl> makes the name scheme safe, as it prevents any duplicate names.
- *owner* is not a CloneZone-specific attribute, it simply makes the entire trigger zone belong to the blue faction, which is important for the next attribute
- *masterOwner* tells the cloner that all spawned groups should be aligned to the same coalition as the zone “*” – which is a short-hand for this trigger zone. Since this trigger zone is aligned with blue (*owner* = *blue*, see previous attribute), all spawned units are aligned with blue. If *masterOwner* was not present, all cloned units would have kept the *template’s* faction alignment.
- *useOffset* again isn’t a CloneZone attribute. It’s is used with any zones that are linked to units. It tells DML that this trigger zone should always follow the linked unit and keep same offset as it is defined in Mission Editor.
- So which unit is this zone linked to? Well, we used ME’s LINKED UNIT interface rather than DML’s *linkUnit* attribute, so it’s not immediately obvious. It’s the blue LUV Tigr, named “B LUV”



8.53.2.2.5 The Replacing/Teleporting Spawner

Name	Value
cloner	
source	Blue Men
nameScheme	<z>-<i>
owner	neutral
masterOwner	*
clone?	replace
useOffset	yes

Now it’s getting neat. If you compare this cloner’s configuration to the previous example’s, you’ll notice that they are essentially the same – except for the nameScheme.

The blue and red spawners used a safe name scheme:

```
<z>-<lcl>
```

That name scheme ensured that each time that a unit was created, it received a unique name (the combination of zone name and local zone count ensures that all names are unique).

The neutral spawner’s name scheme, however, is subtly different:

<z>-<i>

Instead of a combination of zone name and local zone count, it uses zone name and a *group-local* count. The result is that every time that the cloner spawns, *it creates the same names for each and every group each time it spawns*. The main effect here is that since the names are always the same, each spawn cycle replaces any previously spawned units – the previously spawned group appears to 'teleport' to a new location. This is the intended effect for that name scheme (for the *unintended* effect, see the discussion, below).

8.53.2.2.6 The defecting stealth fighter

After we have gone through the previous example, making a plane 'defect' to another coalition seems pedestrian in comparison. All we do here is use two separate spawners that use the same *source* template ("Defector Template") and *identical=yes* attribute, but belong to different factions.

Name	Value
cloner	
source	Defector Template
owner	red
masterOwner	*
identical	yes

Name	Value
cloner	
source	Defector Template
owner	blue
masterOwner	*
identical	yes

Since we use identical=true, any new spawn will replace an existing clone. Since the spawners belong to opposite factions, we can spawn the same unit for both sides. And it is the same unit (as seen from the mission), because they spawn with identical names and ID as the template. That is the reason why both red and blue clones are able to trigger the ME trigger condition for DEFECTOR GROUP DETECTOR.

8.53.3 Discussion

8.53.3.1.1 The missing soldier mystery

The neutral spawning Tigr demonstrated how we can use nameScheme to make groups teleport around the map, and be instantly replaced by a 'fresh' copy.



That name scheme, however, also has **unintended consequences** – and these can be exceedingly difficult to trace. As we had noticed before, the neutral spawns are short one infantry unit. The template 'Blue Men' consists of three infantry and a static object. The cloned result only shows two infantry and a static object.

So where is the missing soldier? Clicking through the spawned units gives us an indication: The two infantry units are named "replacer-2" and "replacer-3". We are missing "replacer-1". Where is it? Clicking on the static object reveals nothing – static units for some reason don't show their callsign.

But it is exactly this static object that is responsible for the missing infantry. Let's walk through the cloner's clone cycle:

- It starts by cloning the group of four infantry. It sets the group count to 1, and generates the names for each unit: "replacer-1", "replacer-2", "replacer-3". Finally, it spawns the entire group into the game, replacing any existing units with the same name.

- The cloner then starts to clone the static object: it resets the group count to 1, and generates a name for the static object according to the name scheme: “replacer-1”. The cloner then spawns this object into the world. Unfortunately, there is already a unit in the world named ‘replacer-1’ – the infantry we just spawned. This infantry unit is immediately removed from the mission, and the static object is spawned instead.

This unintentional side effect is very hard to detect and demonstrates both the power and potential danger of name schemes. To help you detect name collisions inside a cloner, turn on that zone’s verbosity, and the cloner can alert you this.

Add an *verbose=true* attribute to the “replacer” trigger zone next to the neutral LUV, and run the mission, then tell it to spawn a replacer clone. Watch for this



and



Note also that a verbose CloneZone can be quite chatty. This is not to annoy you, but because they are so flexible that is often makes sense to track their work a little closer.

Make sure that you remove all verbose attributes before you ship your mission.

8.54I say hello goodbye (Valet)

8.54.1 Demonstration Goals



Valet is a tremendously useful tool to add player-responsive zones to airfields, helicopter pads, target zones etc. and to greet or send off players. This mission puts it through its paces to give you a general idea of what you can use it for, and just how versatile it is.

8.54.2 What To Explore

8.54.2.1 In Mission

Start the mission, join the blue coalition, and choose one of the two ‘Parker’ Frogfoots.

You enter its cockpit, engines running on the tarmac in Batumi. Immediately you are greeted by the mission:

Good morning, New callsign, welcome to Batumi Center! Your Su-25T is ready to go.

Note that the message mentions you by your current callsign (it’s set to ‘New Callsign’ on my computer, you’ll see whatever it is configured to in your DCS), and the aircraft type is correctly identified as “Su-25T”.

Advance the throttles until you start rolling towards the runway. After a few yards, a new message appears:

Parker-1-1, contact Batumi Tower at 131.0 MHz

This time note that you are correctly addressed by your unit’s callsign, “Parker-1-1”

Taxi to the runway. Take off, and continue on the runway’s heading until another message appears:

Parker-1-1, frequency change approved, have a nice day!

Now turn your plane around and prepare for an approach to Batumi. When you get close to Batumi, another message appears:

Parker-1-1, contact Batumi Tower at 131.0 MHz

Now switch to the “Frog RWY” unit. This time you start on the runway, and you see no greeting message. Take off, continue runway heading and after a short while you’ll receive the same message as when you departed Batumi, except your unit’s name has been exchanged for your current one.

Frogger-1, frequency change approved, have a nice day!

Now switch to the “Blue Approach Tarawa”. Below and in front of you, there are two naval units. Approach them, and soon you receive the following message:

Blue Frog, contact Tarawa on 127.5 MHz for approach

Now exit the cockpit, and switch to RED coalition and enter the one Frogfoot that is available. You again start in the air above and south of the two naval units. Approach, and this time a very different message comes up:

Contact Su-25T, you are approaching a restricted zone. Do not approach or you will be interdicted

If you change to F10 Map view, you’ll also notice that a new unit has appeared on the Tarawa, an AV-8B Harrier that is currently getting ready to intercept you. If you remain in the area, it will most likely shoot you down.

Now change back to blue coalition and enter “Frogger X Target”. Continue on the current heading, and note that you fly over the derelict airfield with crossed runways south of Kobuleti. Note also that you are receiving no messages.

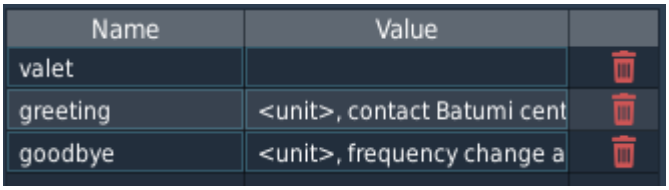
Now enter into “A-10 X-Target” (note: this requires that you have access to the A-10A module). Even before you overfly the airfield, a message comes up

Incoming A-10A, you are approaching target area. Weapons free authorized

8.54.2.2 ME

The Minimal Valet

We start with the minimal setup – the Batumi center. This valet simply greets and goodbyes any player unit that enters or leaves the zone.



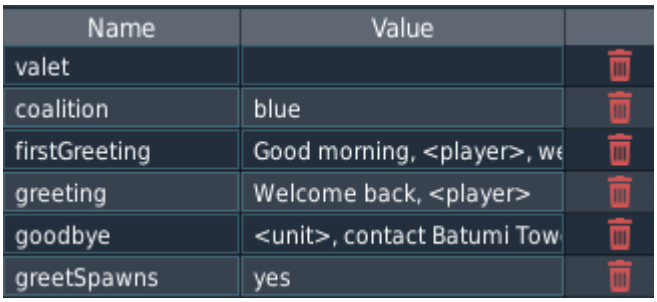
Both greeting and goodbye messages use the <unit> wildcard so that the message is personalized.

Note that if a player’s unit spawns inside this valet, no greeting message is sent to the player because greetSpawns is off by default.

This makes the “Minimal Valet” a perfect greet/goodbye ‘robot’ for your airfields, FARPS or other locations (e.g. hospitals for helicopters)

The Spawn-Point Valet

This configuration of the valet is perfect for all player starting locations. It reacts to aircraft spawning inside it, and when a unit spawns into the valet zone, the initial message is different to the one the player receives should they return later (e.g. for re-fueling).

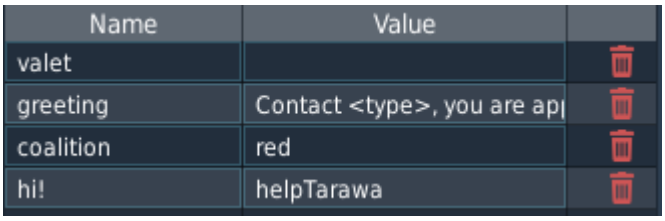


The example to the right is also limited only to blue players (coalition = blue), which usually only makes sense when your airfield has spawn points for red and blue players (big missions with SSB or other slot-blocking logic in place) and you want different messages for red an blue players.

Since this valet greets units that spawn inside (*greetSpawn* = true) its zone, and *firstGreeting* is given, players that spawn into this zone immediately see *firstGreeting*.

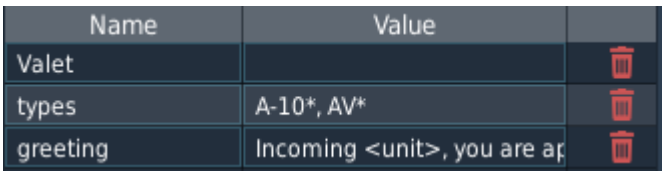
The Valet that triggers

There are two different valets that are linked to the Tarawa and follow it around. Configured differently for blue and red, the red one also uses an important ability. When you approached the Tarawa in a red aircraft, the greeting message was a warning, directed at the individual (the warning uses <type> to correctly address the intruder). The "hi!" output is used here to activate a dormant AI Harrier on the Tarawa that has orders to attack the Su-25T. Usually, you would have more involved cloners trigger here to provide air cover, but this simple example should suffice how the 'hi!' output can be used to trigger events when an aircraft enters a valet zone.



Using the Bouncer: Only some players are admitted

Let's go back to the example where you first crossed the "X" derelict airfield in the SU-25T, and nothing happened, while when you did the same in the A-10, you were greeted by the valet. This is accomplished with the '*types'* attribute. As you can see, you can allow multiple unit types to be greeted by the valet. And since the Su-25T isn't on the list, the valet will not talk to it.



More importantly, though, above types example illustrates an important ability that also extends to the other restrictive attributes groups and units: the ability to wildcard by placing an asterisk "*" at the end of the defining strings. Above type definition greets all players who are flying an aircraft with a type that begins with either "A-10" or "AV". So, currently, this would include the A-10A, A-10C, A-10C_2 and AV8BNA.

8.54.3 Discussion

Nested Valets

Valets can be nested, and the demo shows this with the Blue Player Parking valet, which is nested inside the Batumi Center Valet.



Valets remember a player's inside/outside status independently of each other and therefore usually react at different times.

When you nest valets for spawn points, it's often advantageous if you make the innermost (and only the innermost) valet (the one above the spawn point) responsive to player spawns. That way, players are greeted (and mission can react to player spawns) and you can initiate a chain of messages/sound effects/events when the players progress from the inside to the outside.

When hello is goodbye

Note that when you start in the "Parking" Frogfoot, you are being greeted by the 'Parking' valet, but not by the Batumi CTR valet although you are spawning into both valet zones simultaneously. This is because only the valet at the parking location has a 'greetSpawns = true" attribute, while the Batumi Center Valet ignores spawned units.

Name	Value
valet	
greeting	<unit>, contact Batumi cent
goodbye	<unit>, frequency change a

Also observe a little dirty trick: when your plane leaves the parking valet, it sends you the message "<unit>, contact Batumi Tower at 131.0 MHz", which happens to be the same greeting that the Batumi Center valet sends to players who enter the center zone. So it looks as if this message was sent from the Center Valet's *greeting*, but it is in reality the parking Valet's goddbye message. You can use this and similar tricks to make transitions between nested valets seem more natural and life-like.

Name	Value
valet	
coalition	blue
firstGreeting	Good morning, <player>, we
greeting	Welcome back, <player>
goodbye	<unit>, contact Batumi Tow
greetSpawns	yes

The effect of Spawning

It is important to remember that when a player spawns anywhere on the map, all their previous locations that other valets are tracking are reset. This means that even if a player respawns to the same unit on the same spawn location, all valet zones have forgotten that the player ever existed and will respond to firstMessage/firstSound attributes.

8.55 Davy Jones' Rocker (ASW)

8.55.1 Demonstration Goals

ASW is a small package of modules that add ASW game mechanics to your missions. In this demo we explore how the various pieces fit together and how little effort it is for mission designers to ASW to your mission.



Note:
Since ASW heavily relies on helicopters, and I'm aware that not all mission designers have access to helicopter modules and the ability to fly them, we'll mostly focus on ASW's passive features, and your ability to believe me when I tell you something will happen when you sit behind the controls of an ASW-capable helicopter

8.55.2 What To Explore

8.55.2.1 In Mission

There is a lot to explore, and for some of that you need helicopter modules (Huey, Ka-50, Hip, HIND) and the A-10A. If you don't have access to these models, you can't follow the directions here, but should still be able to understand what is going on.

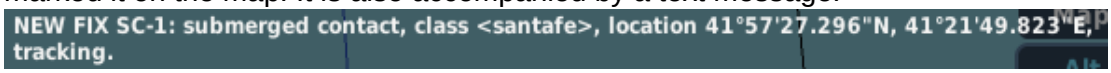
Start the mission in the Su-25T "Frogger", and switch to F-10 map. Then

- Zoom out until you can see the naval units to the west near the coast of Kobuleti.
- Zoom slightly in on the Ticonderoga so you get a better view of the action.
- To the North, two airborne units (Uzi11 and Springfield11) are moving east to west, away from the coast.
- Notice that they leave circular marks behind, in different intervals. These are buoys, dropped by the AI planes

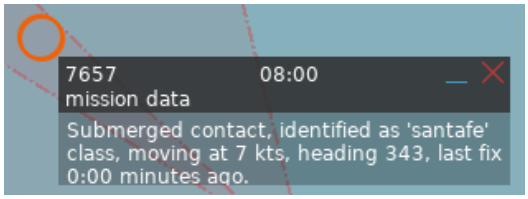


- Occasionally you'll hear a double-beep, and whenever you hear that double-beep, red semi-transparent wedges appear on the map. This is a buoy 'ping' response, alerting the player that buoys have received a signa, the wedge is the 'confidence cone' in which, with a probability of 90% the sub is located.
- With time, more and more buoys are dropped, and the map starts to get saturated with wedges that are all pointing in the general direction of the red sub.

- Soon you'll hear a sound that is like an "U-Boat" sonar ping as seen (or heard) in movies. This announces that fleet has identified a sub, got a position fix, and has marked it on the map. It is also accompanied by a text message:



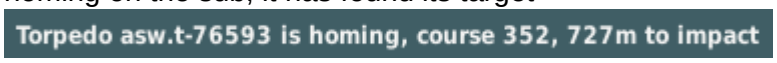
- Note that a ring mark has appeared around the red ARA Santa Fe that is squarely in the middle of the blue naval group. This is the sub that has been 'made' and the ring is the mark. Note that in a normal mission you should disable F10 view options to see enemy units to make detection of subs less trivial.



- Go to Communications→Other→Order ASW Drops and choose "Launch Torpedo T1". The Ticonderoga will launch a torpedo towards the red sub that has been marked.



  The torpedo is marked on the map by a circle that is updated every 10 seconds.
- If all goes well, after a some seeking, the announcement is made that the torpedo is homing on the sub, it has found its target



- Soon after that, the red sub is destroyed:



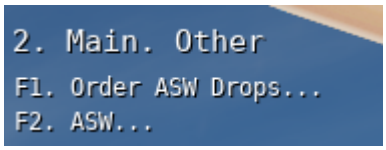
So this was exciting. Make sure that the mission time is less than 3 minutes and that the Stennis is alive and well somewhat NE of the naval group. When in doubt, simply restart the mission. Have the mission time advance towards the 08:03:30 mark, and view CVN-74 Stennis.

- At around 08:03:50 a warning message is displayed:



- A few seconds later, Stennis is hit by a salvo of torpedoes that in all likelihood destroys it.

Now, restart the mission, and enter the 'Frogger' Su-25T. Inspect your Communications→Other… menu. You see two menu items: "Order ASW Drops" (which we have used before to drop the torpedo from the AI Ticonderoga), and an ASW menu. This is the 'real' ASW menu, as supplied by the aswGUI module. Note that this is unexpected since we are not in an ASW helicopter, and that usually, this menu item should not be available to us in the Su-25T. Let's also be happy that we can try this feature in our free plane.

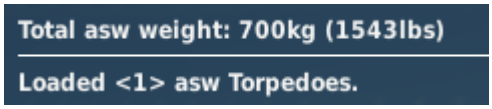


Choose the "ASW…" item.

You are presented with the choice to load either 10 ASW Buoys or 1 ASW Torpedo. A third option (that has no effect if you choose it) gives you an overview of all currently loaded ASW munitions: 0 Buoys and 0 Torpedoes.



Choose to load One Torpedo. This is acknowledged by two new lines of text:



Note the amount of cargo weight, and check that against your plane's cargo capacity.

Now call up Communications→ Other→ ASW up again. Note that the menu layout has changed. You now have the option to unload the torpedo, plus you are informed that there are no more torpedoes in store to load at this location. Also note that your on-board inventory has been updated to show one torpedo loaded.



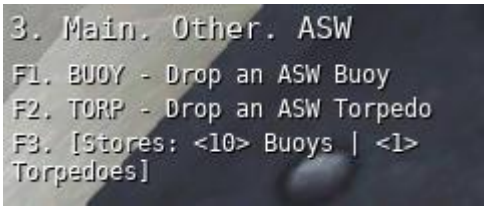
Unload the torpedo, then call up Communications→ Other→ ASW once more time. Note that your stores are empty again, and that a torpedo is now back in store and available.

Now, (if you have access to that module) enter the A-10A or Ka-50. Go to Communications→Other. Note that there is no ASW menu available.

Now (if you have access to that module) enter the Huey on the Tarawa. Use the communications menu to load 4 torpedoes. Then take off and note that you can't get very far before your Huey takes a drink. Note that it is easy for players to overload their aircraft with ASW munitions.

Change into the Hip and load up four torpedoes. Take off from the Tarawa and laugh at the puny Huey as you pass over its watery grave. Touch down again, unload two torpedoes, and load 10 ASW Buoys instead. Depart from Tarawa.

While airborne again, call up communications→ Other →ASW and note that the menu has changed from ASW Supply to ASW Operations. Drop a couple of buoys and perhaps a torpedo or two (even though they are unlikely to hit anything unless you go and hunt for the sub that sinks Stennis).



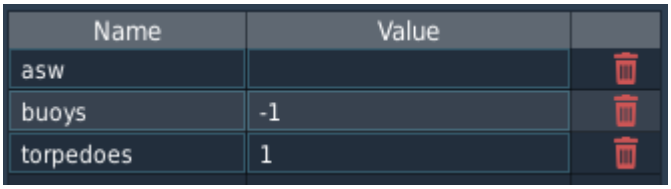
Note how your Hip gets progressively lighter while you drop ASW munitions.

8.55.2.2 ME

ASW Zones: Supply

aswZones serve two purposes: they provide ASW munitions inventory for player aircraft, and can be used to drop ASW munitions by AI-controlled units and free-standing zones. aswZones provide full stock keeping automatically, so if store capacity is limited, loading ASW munitions onto a player plane or dropping them decreases their stock, while unloading ASW from a player aircraft restocks them.

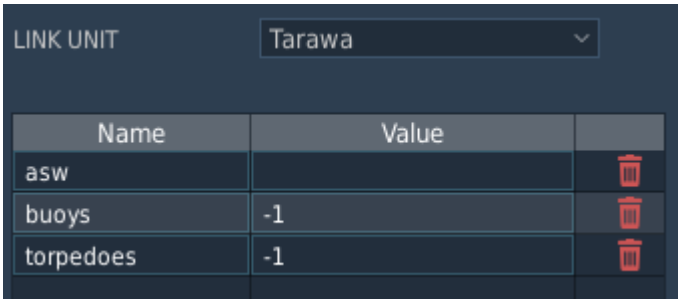
So, let's start with the aswZone in Kobuleti. This example serves multiple purposes. First, it's a reminder that aswZones can be placed on land, so aircraft can not only use land-based



airfields as their ASW base of operations, it can also be used as a way to set up an ASW supply point that (when *real* player-controlled transport planes arrive) can be used to ferry ASW munitions to a naval unit to restock their stores.

Our example here shows that buoys have an unlimited supply (*buoys* = -1), while the torpedo stores are limited - there is only a single torpedo left. In our demo with the Su-25T we loaded and unloaded that torpedo to drain and refill that ASW store.

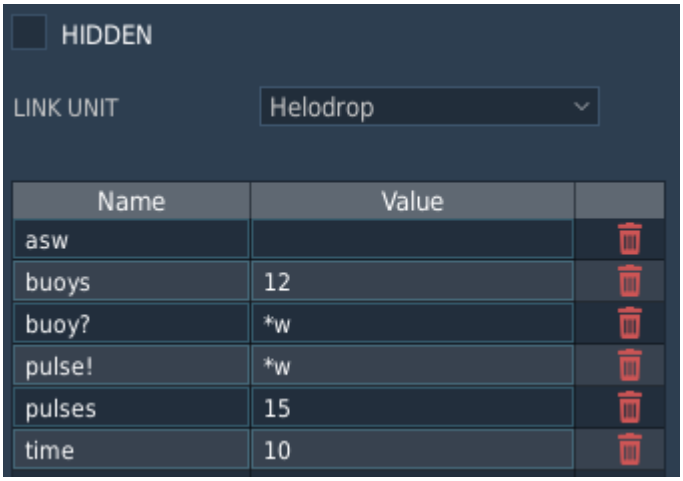
More often, though, you'll want a naval unit to restock your ASW aircraft. The aswZone placed over the Tarawa is the typical example. It shows that stocks for both torpedoes and buoys is unlimited. More importantly, though, is the fact that the aswZone is linked to the Tarawa, so it always follows the Tarawa around.



aswZones: dropping ASW munitions

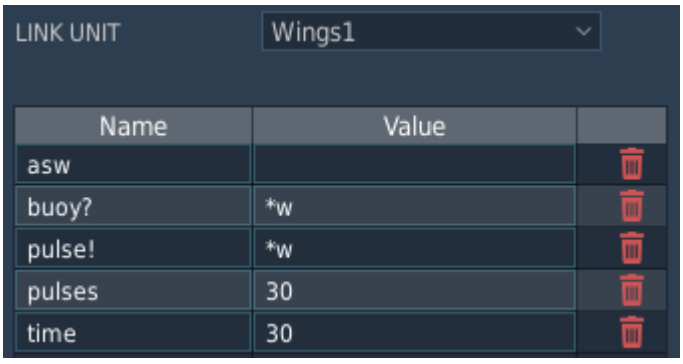
aswZones supply ASW munitions, and they can also be commanded to drop buoys or torpedoes by connecting a flag to their *buoy?* and *torpedo?* inputs.

When used this way, an aswZone drops a buoy or torpedo as long as there are enough of them in store. This way you can easily attach an aswZone to a ship or aircraft, and have them drop buoys and torpedoes on demand. The demo mission does this with the C-130 and CH-53 AI units the fly from east to west and drop the many buoys that help to get a fix on the red submarine.



The example on the right stacks a pulser on top of the aswZone that follows the CH-53 around. The pulser creates a total of 15 pulses on (local) *w (*pulse!* = *w), which feeds into aswZone's *buoy?*, causing the zone to drop a buoy each time a pulse is created. A pulse is created every 10 seconds (*time* = 10) for *pulses* = 15.

Except that this aswZone only stocks 12 buoys. If you count the number of buoys that are dropped, you'll see that even though there are 15 pulses, only 12 buoys are dropped. This shows that aswZones keep track of their inventory, so you don't have to yourself. This means that you could have omitted the '*pulses'* attribute for this pulser, as no buoys are dropped after the 12th.



Also note that this zone uses LINK UNIT to Helodrop (the CH-53), which is the reason why we don't have to provide a coalition attribute for the aswZone: it knows that it is linked to a

unit and automatically fetches the coalition that the dropped buoys and torpedoes belong to from the unit it is linked to.

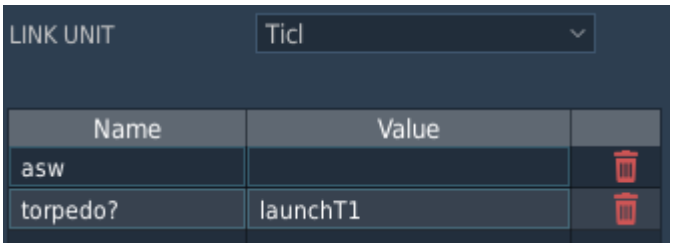
The C-130 AI dropper is set up similarly, except its stores are unlimited, and therefore would endlessly drop buoys if there was no limit on the number of pulses. That is why it needs the *pulses* = 30 attribute.

Note how both stacks use the local flag *w to pass flag information between modules in-stack and without affecting the other stack or polluting the mission's flag namespace.



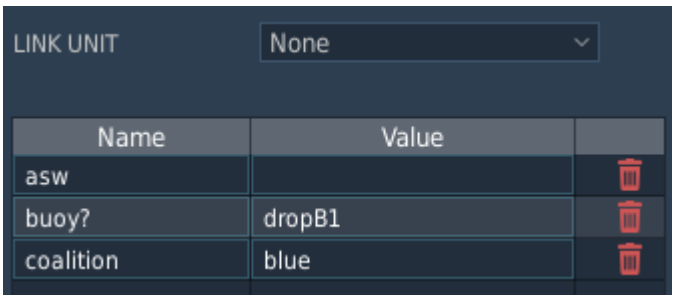
The aswZone attached to the Ticonderoga shows one of the simplest, and incredibly convenient use cases for ASW mission setups:

The aswZone is linked to the naval unit "Ticl", which makes it follow it over the map.



There is a flag 'launchT1' connected to the *torpedo?* input – and that's it. This means that every time that the flag 'launchT1' changes, the aswZone attached to the Ticl drops a torpedo into the water (since there is no torpedoes attribute, this zone has unlimited stores, so it can't run out). If you now set up a radioMenu to provide an output to launchT1, your players can command that ship to launch a torpedo whenever they want.

There is a third use case in this mission that I provide for mere completeness, since it's very rare that you will use it: the unlinked aswZone dropper.



This case is special because unless you place an aswZone on land, you would rarely find use for a stationary aswZone over water, and therefore all aswZones that drop ASW munitions are usually linked to a unit. Should you ever run into a situation where you need to place an unlinked aswZone over water, and this

aswZone should drop buoys or torpedoes, you must also provide a *coalition* that tells the aswZone which side owns the buoys or torpedoes that are being dropped. Here we set *coalition* to blue so the buoy tracks red submarines

aswSubs attack mode



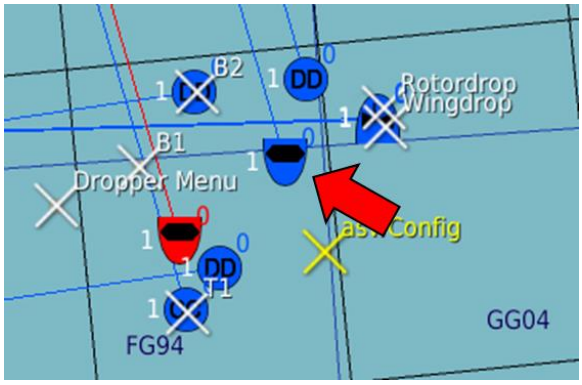
The second red submarine further up north attacks and destroys Stennis. It does so when it gets into critical range of any naval unit that is listed in aswSub's config zone. The attack mode that submarines go into is extremely simple and effective: all they must do is get into critical range (by default 4km), and they attack any ship that is inside a group listed in *targets*. aswSubs does not change a sub's path, so it is the mission designer's job to make sure that subs get close enough. But if they manage to get close enough, they will wreak havoc on any ship that gets too close. In our demo, we set the group "sten" as *targets*, and the *salvoSize* at a fixed 4 (*salvosize* = 4-4), which is enough to kill even a carrier.

aswSubs is a simplistic 'attack dog' module without much finesse. It exists primarily to create believably dangerous submerged threats, and it adds no intelligence to what little exists in the game right now. Most importantly, submarines of opposing coalitions do not attack each other, so you should not expect a fleet of subs be able to fend off enemy subs – that's what the ASW is for.

8.55.3 Discussion

There are a couple of things that are easy to miss:

That blue sub



Have you note that the blue sub mixed in with the other blue naval units? There is an important fact of what is not happening: it is never detected by the blue buoys. That is because aswZones assumes that a coalition always knows where its assets are and the signals for blue subs are filtered. Accordingly, there is never a fix for a blue sub shown to the blue side, to avoid confusion.

Note that this does not mean that torpedoes dropped by blue cannot accidentally kill a blue submarine. Torpedoes attack the closest fix made by same-side buoys, and failing that, the closest submarine that they can sense. And if that happens to be an unlucky sub from the same side – the torpedo doesn't know nor care.

aswCarriers with an *

Name	Value	
aswCarriers	Su-25T, UH-1*, Mi*	
verbose	yes	

We saw at the very beginning that not the default set of helicopters (Hip, Huey and Hind) are able to carry ASW munitions, but so is the Frogfoot (Su-25T). This is because we added the aswCarriers attribute to aswGUI's config zone. This allows other aircraft than the originally intended transport helicopters to access ASW capabilities. This particular config zonbe merely demonstrates an ability to also use wildcard types similar to heloTroops and csarManager: the types listed here are the Su-25T (which allows the Frogfoot access to ASW functions). "UH-1*" means that any user-controlled aircraft type that starts with "UH-1" is allowed access to the ASW menus – which is currently just the player Huey UH-1H. Finally, "Mi*" allows any aircraft type that starts with "Mi" access to ASW menus. And those are the Hip (Mi-8) and Hind (Mi-24).

Challenges

So, how about modifying the demo as follows:

- Allow all Gazelles access to ASW menus



8.56 Taxi Police (policing airfield speed limitations)

Mission designer beware: using TaxiPolice in a mission may decrease its popularity and turn players away from your server: you are imposing some strict rules on players while they are taxiing. *Do not use this module lightly*, only as a last resort when reckless behavior from some players threatens the majority's enjoyment, e.g. during tournaments or re-enactments of a classic mission. Otherwise, trying to impose your will on other people usually is a bad idea – especially when it concerns trifling things.



Usually, on non-strict mission profiles, users should be left to their own devices how they enjoy DCS, and that *includes* taking off from taxiways.

8.56.1 Demonstration Goals

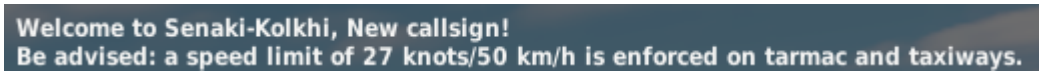
In this demo we explore how the taxi way police can prevent players from exceeding a speed limit on the tarmac, ramp or taxi ways, how we can call the cops and send them home again, and how we can exclude some airfields from authority's scrutiny.

8.56.2 What To Explore

8.56.2.1 In Mission

Start the mission and jump into the Frogfoot on the ground in Senaki ("Senaki Frogger").

Note the greeting you receive as soon as you enter the cockpit:



Since the Frogfoot is pointed invitingly down the taxiway, throttle up and break the speed limit. Soon you'll see



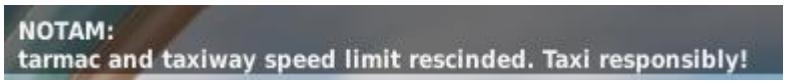
Switch slots to the Frogfoot at Kobuleti (“Kobu Frogger”), and do the same again. Notice that the violation count increases even though you switched plane and airfield.

Now switch to the Frogfoot on final at Kobuleti, and land it. Notice that no speed limit exists on the runway. Turn off the runway (or re-slot to the Kobuleti Frogger on the Taxiway) and break the speed limit another two times outside the runway areas. Your fourth violation ends in retaliation:



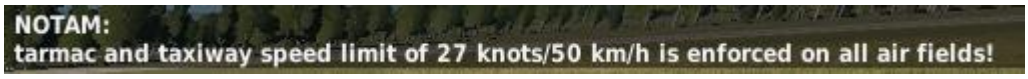
Your plane has become unflyable due to overweight. This will not prevent you from re-slotting, but from now on, every time that you break the speed limit at any airfield, you will immediately be prevented from taking off with that aircraft until you re-slot.

Now, choose communications → Other → 911 Taxi Police and choose “Never mind”.



Now break the speed limit on the taxi way again. This time, nothing happens.

Choose communications → Other → 911 Taxi Police and choose “Call the Cops”



Overspeed again, and get instantly reprimanded.

Switch to the Unlimited Frogger at Kutaisi. Upon entering, note that



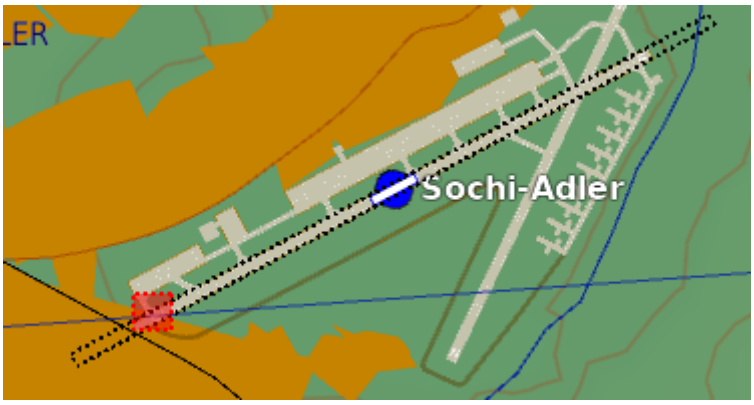
Overspeed on the runway – nothing happens. Freedom!

Now, if you have the module, change into the Huey at Senaki.

Note that you receive no warnings. TaxiPolice ignores all helicopters. So yes. Theoretically, you could overspeed on the taxiway in your rolling Hip, Hind or Apache. But given the fact that it’s so much easier to simply lift off, the likelihood of this happening isn’t worth the effort checking (have you ever tried to wheel-drive a Hip at over 30 km/h while not on a runway?)

Now, again if you have the module, slot into the F/A-18 on the Stennis. Note again that you’ll receive no Police greeting. This is because TaxiPolice doesn’t monitor FARPs nor Ships – for obvious reasons.

Finally, switch to F-10 Map view, and look at the Sochi Adler airfield.



Notice the one dotted black outline on RWY 06-24, and – more to the point – that there is no outline around RWY 03-21. Note also the red square at the threshold of RWY 06.

8.56.2.2 ME
There are few surprises here.

Name	Value
taxiPolice	no

First, Kutaisi was made exempt from any speed limits by placing a trigger zone near the airfield with the *taxiPolice = false* attribute. This eliminates Kutaisi from all monitoring. Note that the trigger zone does not mention the name of the airfield – **it is associated to Kutaisi simply by proximity**. If you moved that trigger zone to a place on the map that is closer to Senaki that to Kutaisi, it becomes associated with Senaki instead.

Name	Value
verbose	yes
onPatrol	anarchy
offDuty	peace

Then, we can enable and disable TaxiPolice by using the appropriate flags. This is set up in the config zone. We use the appropriately-named flags “anarachy” and “peache”, respectively. Also note that we have turned on verbosity, this – among more detailed debugging information – shows the black/red dotted outlines on the F10 map. If you create a mission while enabling verbosity, remember to turn it off before you publish it.

Name	Value
radioMenu	911 Taxi Police
itemA	Call the Cops!
A!	anarchy
itemB	Never Mind
B!	peace

The readioMenu module used to turn TaxiPolice on an off also bears no surprises, providing straightforward access. It’s included here merely to show just how simple it is to expose DML’s deeper functionality to players.

8.56.3 Discussion

When you turn on TaxiPolice's verbosity, it shows black dotted outlines for those zones on airfields where high ground speed is allowed (runways). You can add or remove extra space to the left and right to runways with the 'leeway' attribute, and likewise, add or remove space in front and behind a runway with the 'extend' attribute. By zooming in on the airfields, you can then inspect your settings to determine if your configuration is working for your mission.

Looking at some bigger airfields, however, you may find that this also shows a potential pitfall for players: depending on the map, not all runways (or what looks like runways) are reported to TaxiPolice as runways. In Caucasus, for example, Sochi Adler has two runways: 06/24 and 03/21. However, only one of them is reported to TaxiPolice by DCS, and hence it only allows take-offs from 06/24.





That this is a map issue (not something created by TaxiPolice) shows an example from the Syria map: here we see that all three of Ramat David's runways are correctly reported to TaxiPolice. So if you use TaxiPolice on a map, be sure to check that all runways are covered correctly, and if not, perhaps alter players to that fact to prevent them from accidentally running into a speed trap.

You may also wonder about the red dotted outline. This shows where – according to the map database – an airfield's position (or center) resides. This is important to remember for a number of reasons: TaxiPolice's radius starts here – and TaxiPolice is active in <radius> distance relative to this point. I also suspect (although this hasn't been confirmed by ED) that this is the point from which all airfield contentions (when a faction tries to capture an airfield) are measured: place boots on the ground inside a 2km radius from this point to capture the airfield.

Challenges

OK, so let's see if you have sufficient control over your airfield police: here are a couple of challenges I want you to solve:

- Easy: Add TaxiPolice to your mission, but have it start disabled.
- Less Easy: Make it so that only privileged players can turn TaxiPolice on or off (hint: find a way to password-protect this function)

8.57 Big Score: MoreScore.miz and LaterScore.miz

8.57.1 Demonstration Goals

The PlayerScore module supports more than the basic scoring ability that was demonstrated in “player score.miz” (see the “Keeping The Score” chapter).

There are additional DML demo missions that focus on PlayerScore’s abilities:

- MoreScore demonstrates individual unit scores, and the ‘feats’ abilities
- LaterScore shows how the ‘deferred’ scoring abilities work, and also demonstrate using kill zones that allow scoring to only happen in dedicated zones. It also shows how PlayerScore can export the current score to a plain text file

8.57.2 What To Explore in MoreScore: Feats

8.57.2.1 In Mission

Note: Some feats demonstrated in this mission are much easier to accomplish if you have access to the Huey module.

Start in the “Tango Chaser” Huey (or A-10/Su-25T if you are skilled enough). Enter the left gunner seat. There are three Infantry soldiers, of which only the middle one is red faction. Beyond the soldier you’ll find a LUV Tigr, also red. Kill the red soldier and the Tigr. Note that while you are awarded the kill for the red soldier immediately,



usually the Tigr will ‘cook off’, awarding you the kill only after a minute or so, after it explodes.



Do not change airframes before your kill is awarded or you will lose the kill and achievement!

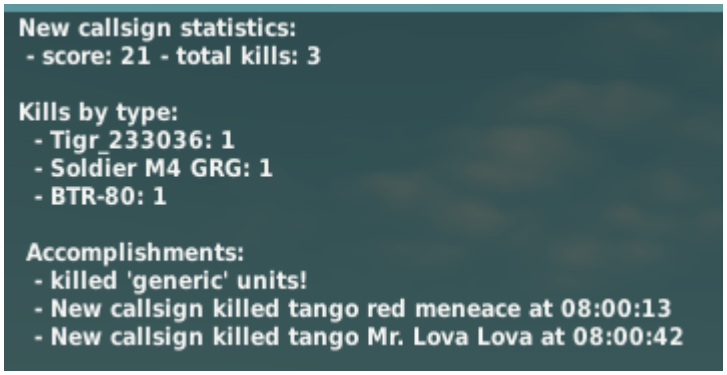
Go to Communication→Other→Show Score / Kills and you should see two kills:



Note that two accomplishments listed below the kill types. Note that the feat describes the unit names (“Mr. Lova Lova” and “red meneace”) as well as the time of kill.

Now take the A-10 or Frogfoot, fly north towards the bridge and kill one of the red vehicles next to it on the western (left) bank of the river, in the general area indicated by the red smoke.

Calling up the score reveals that you received another accomplishment: one for killing a ‘generic’ unit.



Now kill another BTR-80 on the western bank. You’ll receive a kill score as before, but calling up the score screen reveals that you did not receive another accomplishment although the score *did* increase.



Next, if you can take out the three central railway bridge parts. You’ll receive score (50, 51 and 52 points) for each, but you won’t receive any feats, nor will they show up in the type list of killed units – scenery items have no type. Don’t worry about munitions, you have an endless supply.

Now turn around, and head back to Senaki, to the area marked with orange smoke. There are two scenery objects, an Antonov (110 points) and a Mi-8 (100 points). If you can, take them out (they are exceedingly hard to kill, and the A-10’s Mavericks do not lock on to them, and the Huey’s guns can’t kill it, even the Hog’s mighty GAU won’t kill the scenery Mi-8…). Note that again, if you are successful, you’ll be awarded a score, but no feats.

To conclude, land in Senaki. If you want a quick fix, simply slot into one of the airframes on APR, or land the Huey on the runway. Each time that you land in Senaki, as long as the landings are more than one minute apart, you’ll receive a new landing feat but no score for the landing.



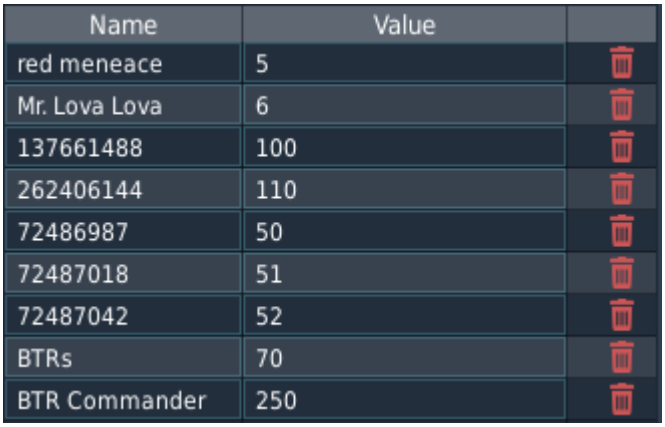
Note that the feat includes the airframe type and time of your touch-down.

8.57.2.2 ME

Score Table

Let's begin with the most complex part of this mission: you may have noted that killing units awards scores that are different from the default score that playerScore awards units. This is because this mission uses a score table in the playerScoreTable zone.

This table defines score overrides for many targets in this mission, so lets's take a look.



Name	Value
red meneace	5
Mr. Lova Lova	6
137661488	100
262406144	110
72486987	50
72487018	51
72487042	52
BTRs	70
BTR Commander	250

The first two entries are 'red meneace' (5 points) and 'Mr. Lova Lova' (6 points), the names of the Tigr (a unit) and Soldier (a static object). Both are name references, and since unit/object names are unique, they each represent single objects.

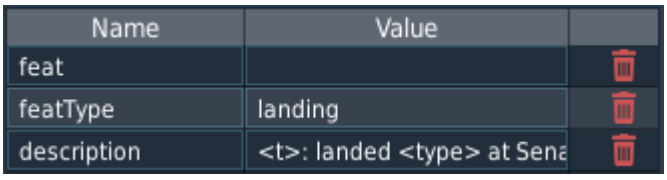
Then there is a long list of mysterious numbers. These are the numbers from scenery objects that we get when we right-click on them and choose 'assign as…'. ME creates a new trigger zone with the scenery object's ID, and we copied that object id as name into the score table. That way you get the correct score when the player destroys the railway bridge elements. The object ids 72486987, 72487018 and 72487042 are the railway bridge elements, 262406144 is the scenery Antonov, and 13766144 is the scenery Mi-8 at Senaki.

WARNING

Object IDs can change between DCS versions, and there is no warning that an object ID has changed. You should avoid using scenery objects as scorable targets for this reason.

Then there are two more entries: BTRs (70 points) – this is the group name of the vehicles east of the bridge. If you hit and instantly destroy one of these vehicles, they will award 70 points – except the one vehicle in that group named "BTR Commander". This vehicle scores 250 points. If the normal vehicles in the BRTs group cook off, they'll only award the default score (because of DCS's static switch-a-roo quirk), the BTR Commander, however, being a named target, will always award 250 points on kill

The Landing Feat



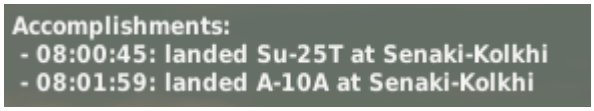
Name	Value
feat	
featType	landing
description	<t>: landed <type> at Sena

Looking at Senaki's runway, you notice that it's enclosed in a trigger zone. This is the feat zone for landings in Senaki. playerScore differentiates between a generic landing score (when you set the default score for landings greater than 0, all landings, no matter where, award that many points and a generic landing feat). Landings inside this zone award a feat. In contrast to playerScore's generic feats, placed feats like these can be curtailed to your missions needs.

You see that featType is 'landing', making it an awardable feat for any player to land inside the zone. The description attribute holds the wildcard-supporting text for the feat that is awarded:

```
<t>: landed <type> at Senaki-Kolkhi
```

Like *messenger* and *valet*, *playerScore* supports a wide variety of wildcards, and in this case, the feat's text is created dynamically when the feat happens.



Above the <t> and <type> wildcards were resolved dynamically to their current values, making the feats player-individual and much more relevant.

Kill Feats
There are two kill feats placed on the map: the 'terrorist feat' and 'kill feat (once)' zones. Let's start with the 'terrorist feat':

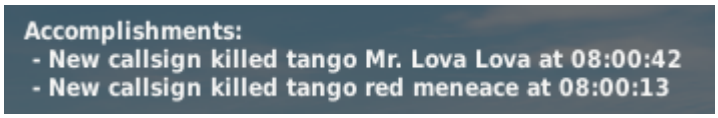
This is a "kill" type feat, meaning that when a kill happens, playerScore checks if the killed unit is inside this zone, and if so, awards this feat.



Again the description accesses context information about this kill via wildcards:

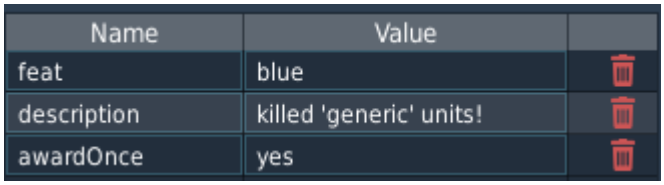
```
<player> killed tango <unit> at <t>
```

which translates at runtime into their real values



The 'kill feat (once) zone is very similar with two subtle differences:

First, we see that there is no "featType" attribute. This is because we simply use 'kill' as default, unless you give another type, playerScore assumes that the feat is a kill feat.

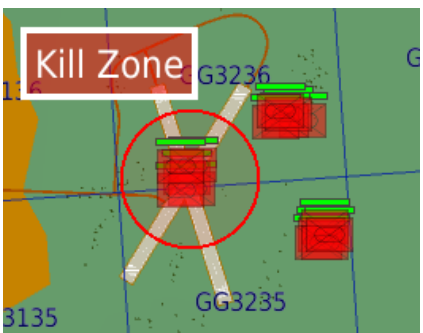


Then, there is an awardOnce attribute. This means that every player can be awarded this feat at most once. That is why, when you killed the second vehicle in that zone you did not receive a feat. Use this option to prevent 'feat-inflation' in your missions.

8.57.3 What To Explore in LaterScore: Deferred Scoring / Kill Zones

8.57.3.1 In Mission

Take command of one of the "Shooter" aircraft (A-10 or Su-25T) inbound on the group of targets south. Stop the



simulation, and look at the F10 map. Notice the “Kill Zone” and units located both inside and outside of it.

First, take out a vehicle that is *not* situated on the runways (i.e. one of the two groups east of the runways).

As soon as you destroy one of them, you’ll receive a notification that a kill was registered but did not count due to it happening outside of any ‘legal’ kill zone.



Now destroy a unit that is located inside the marked kill zone. Note that after the first unit is hit, units try and disperse, and some of the units that were formerly inside the kill zone may have moved outside. Make sure that the unit that you are targeting is squarely inside the zone. Upon destruction, you’ll receive something like this:



Note the ‘awarded after landing’ bit. Check your current score with Communication→Other→Show Score



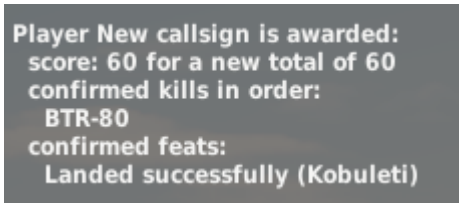
Indeed, your score is currently still zero, with 10 points ‘unclaimed’. Now land your aircraft in Kobuleti (and only Kobuleti). You must land successfully. Upon touch-down, you’ll see the following notices appear in rapid succession while your plane is still moving:







Slow down and remain on the runway for the next 10 seconds until you see the following appear:



The unclaimed score is now awarded to you, and you also have received a ‘landing’ feat that first (for all of 10 seconds) was withheld, and after the 10 second wait was awarded with the other pending scores.

Now take off or re-slot in the airframe attacking the kill zone, and kill another unit inside the kill zone (i.e. so you have 10 unclaimed points). Now, after seeing your score being held, crash your aircraft, or re-slot. Note the messages that appear:

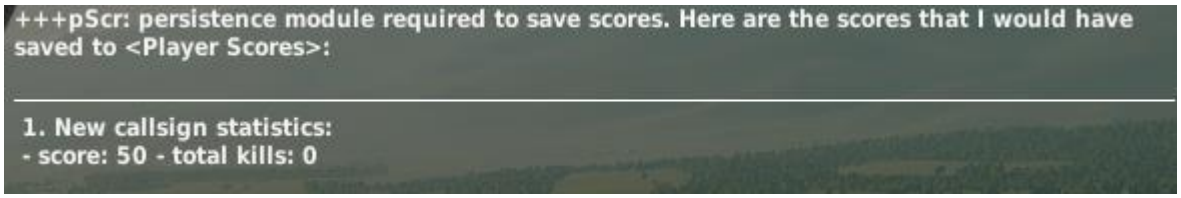


Scores are kept on an individual basis (not aircraft), but as soon as you enter a new (or re-spawned) aircraft, all pending points and feats are discarded.

Restart the mission (to reset all scores), and choose one of the 'Lander' aircraft for Senaki (if you are lazy, jump into the Huey in Senaki), and perform a couple of landing (if you are controlling the Huey, note that landings have to happen after some time interval in order to count).

Note that landing in Senaki will add score and feats that are queued for you to receive in the future, but not upon landing in Senaki. If you want to claim those feats, head over to Kobuleti and land there. And as you have seen before, simply switching to an aircraft on approach in Kobuleti will not help you, all feats achieved in Senaki are discarded.

Finally, choose Communications→Other→Save Player Scores→Save Scores. You'll see something like this:

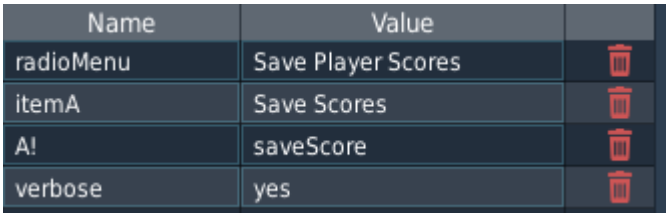


No score file is created on your PC because the 'persistence' module is not included in the mission, and I'm assuming that your DCS version isn't de-sanitized. But PlayerScore still tells you what it would have saved. Note that there is a ranking, and only the score summary would have been exported to text file had persistence been included and your DCS instance been de-sanitized.

8.57.3.2 ME

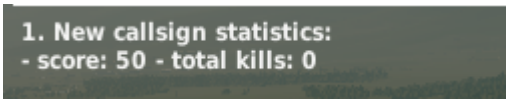
Saving Score to Storage

Let's begin with saving score to file. This is only 'simulated' in this demo, as it requires the persistence module to be included in the mission (which I intentionally excluded to show that PlayerScore still works), and that the DCS instance running be de-sanitized (see the persistence chapter on what that means).



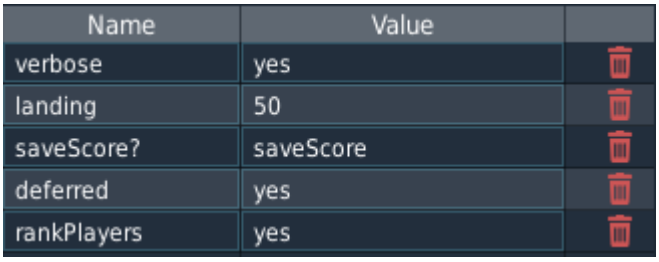
Enabling PlayerScore to save the current score to file simply requires a flag to change that is connected to the input saveScore? – and we do that with a radioMenu item for simplicity. Here we use the aptly named flag saveScore, so every time you choose that menu item, it will cause PlayerScore to try and save the current score table to storage (and fail non-catastrophically, for the reasons explained above)

The config zone contains another entry *rankPlayers* that is relevant for saving scores to file: when this attribute is set to true like here, the players are ranked by their score, and the rank is evident in the list. If you leave rankPlayers to default (false), the scored table will not issue rankings, and list players randomly (or rather: as they are stored in PlayerScore's internal list, which appears to be random to most people). Finally, there is no *scoreOnly* attribute in the config zone, meaning that it defaults to true – that is why the score table is so concise and omits all feats.



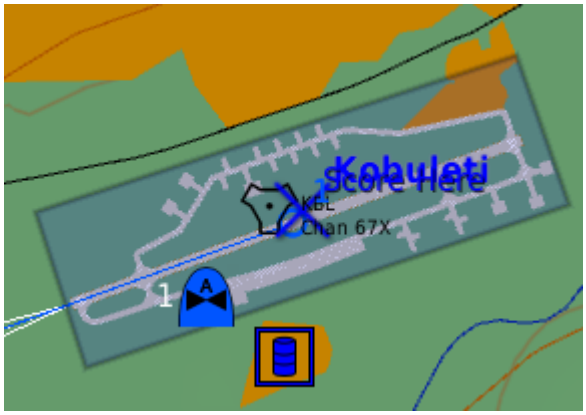
Deferred Scoring

Next, we see that each landing awards 50 points – and automatically a generic landing feat as well. The most important setting, however is *deferred* set to true. This is what enabled the 'award score and feats after landing' mechanic.



Score Safe Zones

In order to work, mission designers must designate zones in which the player must land in order to claim their score and feats. Be careful when placing this zone, as the landing event must happen inside this zone. For helicopters this is usually not an issue, but for fixed-wing aircraft this means that the zone must include a runway, as most planes usually tend to not survive touch-downs outside runways (I hear you Harrier pilots snickering in the background!). In our demo mission we only put the Kobuleti airfield area inside a zone with a *scoreSafe* attribute, making landings in this zone the only place on the map that awards the accumulated score and feats.

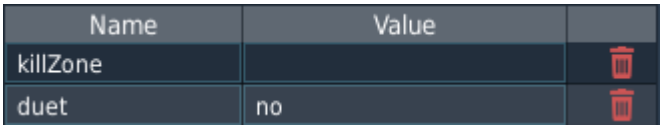


That is also the reason why landing in Senaki does award score and a feat, but those are only awarded to the player after they land in Kobuleti. After landing, a default waiting period to remain inside the scoreSafe zone is required (10 seconds), which again is set in the config zone (no entry = default 10 seconds).



Kill Zone / Duet

The presence of a single kill zone on the entire map switches PlayerScore to 'killzone only' mode: kills and kill-related feats only score when they occur inside a kill zone. Somewhat uncharacteristically, I added the *duet* attribute even though adding this attribute is only required should you want to set it to true. When *duet* is disabled (as it is here) only the destroyed unit must be inside a kill zone in order to score. That is why you can kill units inside the kill zone with stand-off weapons like Mavericks. If *duet* is enabled, both units (player and killed unit) must be inside the *same* zone at the point of kill.



8.57.4 Discussion

Although PlayerScore is (supposedly) a simple droop-in module, there have surfaced a lot of points while I wrote the documentation that may merit some additional thought or highlight. Here they are:

PlayerScore and PlayerScoreUI

PlayerScoreUI has a single function, and that is give each player access to their individual score. For PlayerScore 2.0, I rewrote PlayerScoreUI to be completely self-sufficient, and very, very lightweight. It no longer requires any other modules to load before it does (not even PlayerScore). If you look at the MoreScore demo, you'll notice that PlayerScoreUI loads before PlayerScore, just to prove this point.

Scenery Object Score

Scenery objects (like some buildings, bridges, etc.) in DCS are quirky: they can be ridiculously difficult to destroy, some are even invulnerable. Some can't be targeted by EO weapons. And they tend to change their 'name' (which is their ObjectID) between DCS releases. As such, they make bad, high maintenence objects for mission designers. Sometimes, however, we must use them (for example when a bridge is our target). If you are making the destruction of some scenery objects scorable, make it a habit to regularly check your mission after releases to see if their ObjectID is still valid.

The "cooking off" issue



A change introduced by ED with DCS 2.7 has some wide-reaching consequences: the 'cooking-off switcheroo'. This can happen when a during combat vehicle isn't killed outright (i.e. damaged 100%) but gradually gets damaged until it dies. Visually, in DCS the vehicle catches fire, and after a while blows up. For (possibly) better performance, when DCS determines that a vehicle is essentially dead and initiates the burning animation (but before it explodes and the 'dead' event occurs), it switches the ground unit (AI-controlled) to a static objects (uncontrolled) of the same type and with the same name. The issue here is that in DCS's architecture, static objects have no group that they can belong to. This becomes only a problem if you are using a Score Table in your mission that bases the score on the group name. If the unit cooks off, the player will only receive the fallback (default) score for ground units instead of the score intended for the group.

Challenge: make not the railway bridge the target but the road bridge

Make a landing feat for a moving aircraft carrier. The landing zone must not be much larger than the carrier itself

8.58 Effects with a Flare.miz (FlareZone)

8.58.1 Demonstration Goals

Fireworks! At least that is what we can create with flare zones. The module is very easy to use, and this demo serves mainly to showcase some of the non-standard, more creative uses that this module offers.



8.58.2 What To Explore

8.58.2.1 In Mission

Enter your “Frogger” Frogfoot. It’s dark. Look out towards the end of the runway, then choose Communications → Other → Flare Control → Launch Flares.

Note that along the runway, on both sides three sets of flares are launched.

- Closest to the aircraft a red (left) and white (right) flare are launched. Both flares are launched almost perpendicular to the runway: Towards North on the left, and South on the right.
- Slightly further away there is again a flare launched per side, but their colors are randomized. They launch away-ish from the runway: in northerly direction on the left, and southern on the right
- Finally, attracting all attention are the colorful launches of 50 to 100 flares on both sides. Again, the flares launch northerly and roughly south on each side

Press F2 (outside view), zoom out and place the camera so that you have a directly overhead view, similar to map view. Go to Communications → Other → Flare Control → Launch Flares again, to launch another volley, and observe the flight paths of the flares. Verify that very few, if any at all, touch the runway, they all launch away from the runway.

And now, purely eye candy: switch the to ‘Showboat’ Frogfoot. Notice that as soon as you enter the cockpit, lights come on, produced by two LUV Tigrs that also have appeared. Now press F2, zoom out slightly, and turn the camera until you are looking directly at the plane’s nose, and are almost even with the plane.

Request Communications → Other → Flare Control → Launch Flares and enjoy the fireworks that are worth of an airshow or aircraft presentation.

8.58.2.2 ME

Let's start with the bare necessities. What I like about flare zones is how simple they are. You need a red flare that launches from the zone's center when the red flag is pulled? Here you go. One (default salvo size) flare is launched North (0°, default direction), from the center of the zone when 'go' changes value.

Name	Value	
flare	red	
launch?	go	

So let's proceed directly to the Big Kahuna:

This flare zone launches

- flares with random color (each flare with random color)
- whenever the flag 'go' changes its value
- in a direction from 90° (East) to 270° (West), with each flare getting an individual direction between 90° and 270°
- Each time the 'go' flag is polled, between 50 and 100 flares are launched.
- The entire launch takes 2 seconds

Name	Value	
flare	rnd	
launch?	go	
direction	90-270	
salvo	50-100	
duration	2	

8.58.3 Discussion

There is one special case to observe. Above we launch flares in a direction between 90 and 270 degrees, meaning that each flare has a heading of 90° or higher, and 270° or lower, making all flares launch in a southern arc. Simple – as this is exactly how DML processes ranges: lower bound (90) to higher bound (270).



There is one problem, though: what if we want to launch flares in a Northern arc? Naively, we'd set a range of 270 to 90. But this will cause DML to exchange lower and higher bounds, and all flares would again be launched in a southern arc.

So, we could try to set a range from -90 to 90. That, unfortunately, doesn't work either because of how DML processes ranges: both bounds must be positive numbers. The somewhat inelegant is a work-around (literally, we go around the circle): we start at 270°, and end at 90° + 360° = 450°.

Name	Value	
flare	rnd	
launch?	go	
direction	270 - 450	
salvo	50-100	
duration	2	

8.59 Players in the Zone.miz (PlayerZone)

8.59.1 Demonstration Goals

This mission is a short example of how you can use playerZone to trigger when players enter and leave a player zone, and use the pNum attribute to get the number of players that are currently in the zone

8.59.2 What To Explore

8.59.2.1 In Mission

Enter one of the aircraft, and continue flying that heading. After a short while, your plane enters the playerZone. A message appear



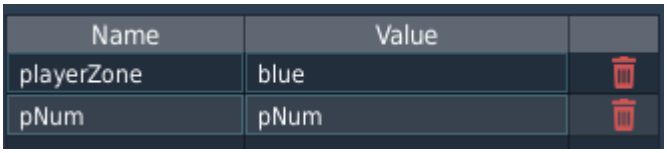
s:

Continue on your heading, perhaps switch to F10 Map view to watch your plane leave the zone (marked as a green circle). As soon as your plane is outside, another message appears:

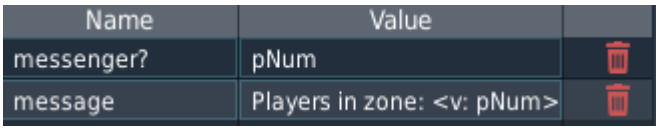


And that's already it.

8.59.2.2 ME



We are using the player zone's 'pNum' Attribute to generate the events that trigger the messenger. Since messenger triggers on change, and the (irresponsibly named) flag "pNum" changes whenever we enter the zone or leave it (because the pNum output carries the number of players in the zone), we do not need the "added!" nor "gone!" outputs. To trigger messenger



Messenger triggers on a change in pNum (the flag that is connected to the output of that name always carries the number of players in the zone) and also references the flag "pNum" in the wildcard <v: pNum> to access and display the current number of players in the zone.

8.59.3 Discussion

Irresponsibly named flags

You should avoid naming flags after one or more of the attributes that it connects. It is tempting, but sooner than later will you run into issues because you will confuse a flag with an attribute.

Triggering on pNum is unsafe.
We are using a lazy trick to simplify using messenger: we use playerZone's "pNum" output from playerZone (with a confusion-inducing flag also named "pNum") to do two things: trigger messenger, and supply the number of players inside the zone.

Although this should work most times, this setup assumes that every time that a player enters or leaves, the counter pNum changes. This should, even in multiplayer, mostly work. But be aware that if, within the sample window (usually 1 second) one player unit leaves the zone, while another enters it, the net result is that pNum remains the same number, an messenger will not be triggered. If this is cause for concern for your mission, you should use the outputs "added!" and "gone!"; those outputs will reliably generate signals, even if the net number of players remains the same.

8.60 Not too shallow at all.miz (Remove ship husks)

8.60.1 Demonstration Goals

Shallows removes destroyed ship hulls that fail to sink because the water is too shallow for them to sink below the waterline (usually inside of harbors). The problem with this is that in DCS there are few ship models that have a ‘destroyed’ version; after the ship sinks a bit, it sits in the harbor, looking undamaged. From a plane’s vantage, these dead ships are indistinguishable from undamaged, alive ones.



Shallows first places a fire animation over a dead ship for five minutes, and then removes the hull and fire effect.

8.60.2 What To Explore

8.60.2.1 In Mission

Enter one of the ‘Looker’ planes to look at the ships at harbor. Then enter one of the attack planes, and sink one or two of the ships. As soon as they are destroyed, return to your “looker” place. Wait and watch as the ships detonate, and then sink… a bit. The harbor is too shallow for them to submerge.

After five minutes, the ships and the fire effect vanish.

8.60.2.2 ME

It just works.

8.61 No Gap, Nop Glory.miz (StopGap, StopGapGUI)

8.61.1 Demonstration Goals (Single/Multiplayer)

StopGap replaces empty player slots on the ground with static aircraft of the same type and livery until a player slot into them. This can be a welcome feature for missions where the designer has prepared many slots to provide a player with many options to choose from, but as a result the airfield looks barren and empty. StopGap populates the airfield with the correct static replacements for a significant increase in eye candy. The price to pay: some performance on the airfield (when going overboard with the types and number of slots). StopGap works in single- and multi-player; in multiplayer, the serving computer should run the provided 'StopGapGUI' module to prevent synchronization issues during spawn-in.



NOTE:
In order to test StopGap's *multiplayer*-ability, you must first install the stopGapGUI server module on the server.

8.61.2 What To Explore

8.61.2.1 In Mission

Enter any of the aircraft that catches your fancy. To keep with DML's tradition of only using freely available aircraft, we use one of the many Frogfoots.

Look outside. Notice the many planes that populate the airfield. Rejoice.

Now choose Communication→Other→StopGap Control→DISABLE StopGap

Look outside. Feel lonely.

Choose Communication→Other→StopGap Contol→Turn StopGap ON

The aircraft return! Now choose any of the other planes that you can see on the airfield and that allow you to change into them (perhaps a Flanker, Hog or Tiger – you need to own the module to do so), or maybe another Frogfoot next to you.

Note that the Frogfoot that you left is replaced by a static once more and you now sit inside the formerly static aircraft and can control it.

Now enter one of the 'Three Kings' A-10A a group of three player-controlled Hogs, preferrable the middle one (if you don't own that module, please read on and simply believe that wat I describe really happens). Once you are in the cockpit, look to your left and right and notice that there are no other Hogs from your group. Only your player A-10 is visible.

[Ensure that you are in single-player mode]

Now enter the Harrier “Harry” or “FARP Hooi” (if you own those modules – if not, please accept the following as gospel). On the Farp should be two aircraft: a Huey and a Harrie, each facing the opposite direction. Enter one of them, and look out to the right. You should not see the other aircraft in single-player mode.

Now restart the mission multiplayer, and re-enter one of the two FARP-base aircraft and look out to the right. You should see the other aircraft (Harrier or Huey).

8.61.2.2 ME
There is very little noteworthy about this except a couple of things:

- It just works out of the box.
- The airfield looks so much better when populated.
- As you would expect, the on/off options for stopGap are provided by a radioMenu module that bangs! the on? and off? inputs set in the configuration zone.
- To exclude units or groups from stopGap (i.e. those planes’ slots aren’t filled with static stand-ins), use trigger zones with the `stopGap = false` attribute
- To exclude units or groups from stopGap in single-player only (but not multiplayer), use a trigger zone with an attribute `stopGapSP = false`
- The “Three Kings” group is a group of three player aircraft which is against stopGap recommendations. See the ‘NoGap’ documentation for how to side-step these-

8.61.3 Discussion

Always think about providing an option for your players to turn stopGap off, perhaps even start the mission with the config zone’s onStart option set to false. This way, if a player’s machine buckles under the strain of too many aircrafts, or when synchronization issues in multiplayer due to unforeseen changes in a future DCS version threaten to make the mission unplayable, they have a quick stopgap (yeah, I know. Puns are us).

Also, the mission uses one `stopGap = false` attribute in the light blue trigger zone in Senaki to hide the second player Su-25T in the ‘Invisible parts’ group. You can use such zones to remove specific units, groups, and areas from stopGap.

Since there are some detrimental side effect in single-player (where the mission cannot take advantage of the superior marshalling capabilities afforded by the stopGapGUI server script) like aircraft falling from great height upon spawning, you can selectively disable stopGap for planes in multiplayer only by placing a trigger zone with a `stopGapSP = false` attribute. Those planes will be stopgapped in multi-player, but not single-player.

The ‘NoGap’ alternative script to ‘stopGap’ can handle multiple player aircraft within the same group, but has other drawbacks instead. Be sure to read up on NoGap and run through the ‘No Gap, No Problem’ documentation to decide which version you want to use in your mission.

8.62 No Gap No, Problem (noGap).miz -- tbc

8.62.1 Demonstration Goals

NoGap functions essentially the same as stopGap, with a single notable difference that is only evident in edge cases: player groups with multiple player-controlled aircraft, and the difference in only evident for those moments when at least one player is slotted in.



This mission demonstrates the subtle difference that NoGap's method to populate airfields has versus stopGap. Be sure to have run the 'No Gap, No Glory' demo mission and read the demo's documentation before you run this one.

8.62.2 What To Explore

8.62.2.1 In Mission

The significant difference between NoGap and stopGap is only visible when you slot into one of the 'Three Kings' A-10A. Slot into the middle (second) A-10A, switch to outside view and zoom out.

You'll note that your Hog is flanked by two A-10A statics. With stopGap, those two aircraft would have disappeared when the first player (you) slotted in. Not so with NoGap.

8.62.2.2 ME

There is no significant departure from the "No Gap, No Glory" mission set up. Read that mission's description.

8.62.3 Discussion

NoGap gives you individual player plane static replacement controls for groups with more than one player aircraft. The drawback here is that NoGap, due to being unit-focused rather than group-focused, is not compatible with popular mission scripts like SSB – because these scripts focus on entire groups.

8.63 My first Factory.miz (OwnedZones 2, Factory 2)

8.63.1 Backstory

The original “ownedZones” module contained the functionality of conquerable zones, plus the ability to produce defensive and offensive units. As of version 2 of owned zones, the latter production functionality was split off into its own ‘factory’ module.

8.63.2 Demonstration Goals

This demo builds on the “Owned Zones ME Integration” demo that introduces the concept of capturable zones and production. When OwnedZones matured to level 2.x, the production ability was split off into its own factoryZone module.



In this demo we look at some of the finer, more advanced details of both modules. It’s important to remember that as of version 2.x, a factoryZone is always an ownedZone - but not vice versa.

8.63.3 What To Explore

8.63.3.1 In Mission

There is a lot happening on this map, with a certain element of randomization, and you may need to run the mission multiple times to catch all the details.

Now run the mission from inside your trusty Frogger (or Hogger). Note the helicopter departing to your left. Almost immediately, you'll see the message: "blue now owns 4 zones".



Switch to F10 map view, and zoom out until you can see the entire airfield, and its surrounding.

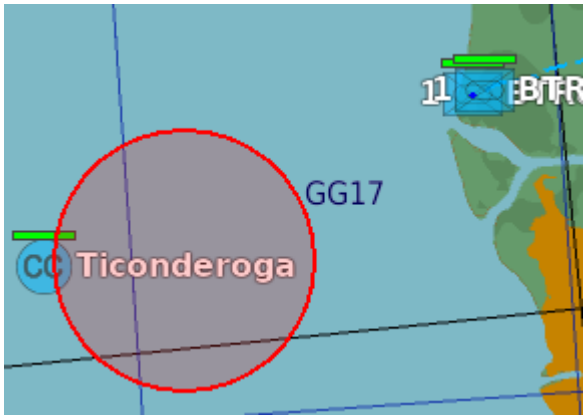
Note the black zone at the end of RWY 09.



Then, try to spot the three factory zones (one red, two blue), easily detectable by the presence of defensive troops, west of Senaki airfield.

There is a fourth blue factory/owned zone, roughly at the same height of Senaki's runway, which does not have defenders, and can be harder to spot. South of runway 09's threshold, there's a quad red owned zone, also empty.

Keep an eye on the departing AH-1W depart, it will soon begin an approach on parking slot 66, which is inside an owned zone

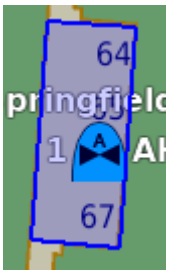


While the helo buzzes around, zoom out and note the red owned zone over deep water, off the coast, west of Senaki. A Ticonderoga class naval unit is approaching that zone, and will take some time to get there.

At the coast, note the units that mark yet another blue factory.

(if you are easily bored, use accelerate time)

At around the 03:20 mark, the Cobra touches down, capturing the neutral zone that existed around that location. A message "BLUEFORCE have secured zone Helo Cap" appears, along with







At 05:00 all factories spawn their first offensive forces, easily visible on the map. The defenseless factory spawns a Leo, all other zones spawn BTR-80 or infantry soldiers.

A few seconds after spawning, all spawned offensive units start moving towards the closest owned zone that does not belong to their faction. At this point in time, the two most interesting units are the blue Leo which moves north, and the red BTR-80 which moves south.

It will take them a few minutes to reach their targets, so look southwest to the water-based owned zone, and especially note the blue BTR-80 that are moving towards the closest non-owned zone. Note that they are **not**, in fact, moving to the closest enemy owned zone - that would be the deep-water red zone. They ignore that zone and move towards the Senaki-Kolkhi-based red factory instead.

Keep your eyes at this region, for at roughly 06:00, the Ticonderoga reaches the red deep water owned zone. and captures it, accompanied by the messages

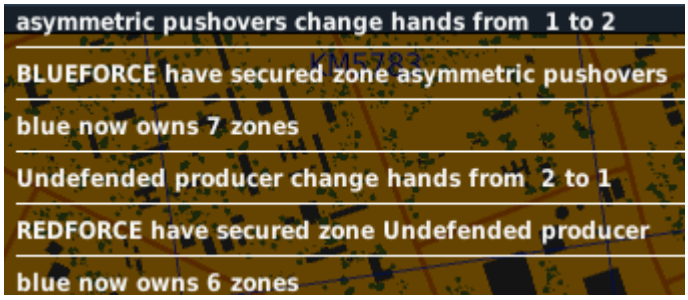


and



Let's bring our focus back to the zones around Senaki. The red BTR-80 makes its way past the blue Leo (>80% chance, it usually only takes a light beating) and both vehicles continue their way towards their destination: the Leo towards the red factory, and the BTR towards the undefended blue.

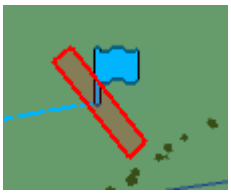
At 08:20 both reach their destinations, each capturing their destination zone. You'll see messages similar to this appear:



Note that the vehicles don't stop, they immediately seek out new targets: the red BTR-80 continues south, while the Leo turns around and heads to the zone it came from.

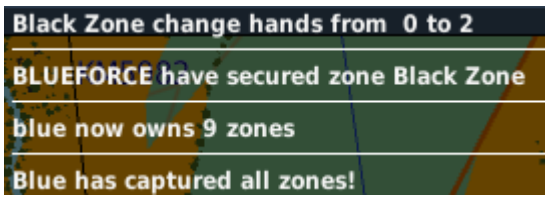
What happens next is completely random, but eventually, the red BTR is destroyed, and all the zones it captured and left behind unprotected are recaptured by blue (run the mission multiple times to perhaps see different outcomes).

There's still one red, quad-shaped owned zone remaining that's the target of at least the M4 soldiers making their way, maybe some more troops. Wait (perhaps in accelerated time) until the last red zone falls, usually around 15:30



All troops then start heading towards the black zone at the end of RWY 09.

At long last, around 22:00, the first blue unit reaches the black zone and conquers it. The following messages appear:



Note also, that instead of turning blue - as you may have expected, the black zone turns into one with a yellow outline and grey fill color.

8.63.3.2 ME

Most of the differences that we are covering here are set up in the config zone for ownedZones, and at the beginning of the mission we already saw two important changes from the 1.x version of owned zones, so let us inspect the config settings for this mission:

- *heloCap* is set to “true”, which allows helicopters to capture an owned zone simply by touching down inside of it. We saw this at 03:20, when the cobra captured the neutral zone at Senaki airfield

Name	Value	
navalCap	yes	
heloCap	yes	
blueOwned#	blues	
allBlue!	blueDone	

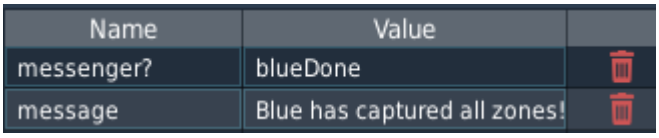
- *navalCap* is also set to true, allowing naval units to capture owned zones. Before, owned zones could only be on land. Now there can not only be water-based owned

zones, the GroundCommander module is smart enough to ignore water-based owned zones when commanding round units to capture them.

- *blueOwned#* always puts the number of zone that blue currently holds on the flag named “blues”. This is fed into a messenger zone that not only triggers on a change in that flag, but also accesses that flag with the `<v:blues>` wildcard to output the number of zones currently held by blue, creating the “blue now holds xx zones” messages that appear each time that blue gains or loses a zone.

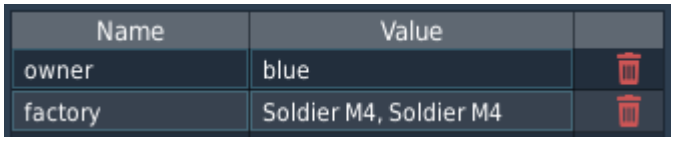


- *allBlue!* is an output that changes the value of flag “blueDone” when blue has captured all owned zones on the map. This simply triggers the messenger connected to “blueDone” to announce that blue has conquered all zones – usually triggering a win condition.



There are a couple of factories on the map that all show some interesting aspects, representing a number of factory ‘archetypes’:

The Basic Factory
First, note the owner = blue attribute, and recall that all factories are also owned zones, and the presence of this attribute ensures this.



The factory attribute specifies 2 “Soldier M4” as types for all production (defensive and offensive). This factory first produces these types as defending units, and after that is done it produces the same as offensive units, regardless which faction this factory belongs to.

The Defended Warehouse
This variant of the basic factory only produces defenders, which is why I call this construct a ‘warehouse’ – it doesn’t produce attackers, but defenders and replenishes them after they have been destroyed. This is accomplished by leaving the default production (*factory* attribute’s value) empty, and providing a *defenders* attribute instead. defendersRED and defendersBLUE both inherit from defenders, so both factions produce the same defenders, and no attacking units.



The Undefended War Factory
Mirroring the “Defended Warehouse” from above (which only produces defensive units), this construct only produces attacking units by leaving the *factory* value empty, and supplying a



production attribute with the types to produce. Here again we use the inheritance ability to set up defenders to none, while productionRED and productionBLUE inherit from production. This factory therefore has no defenders, and produces Leopard MBTs as offensive units

The Asymmetric Factory

This variant produces different types for different uses (offensive and defensive) and different factions. This example is but one of many possible, and it merely serves to show the flexibility of the production inheritance mechanism.



Here, the factory by default produces BTR-80 for defensive and attacking units. Unless red owns this factory, where it still produces BTR-80 as attack units but is defended by a Soldier M4. You can use this to great effect to tweak a mission's difficulty. In this case, it would be easier for blue to capture the factory (red defends the factory with an Infantry soldier) than for red (blue defends the factory with a BTR-80)

8.63.4 Discussion

The initial "blue now has 4 zones"
When the mission starts up, there is the initial message



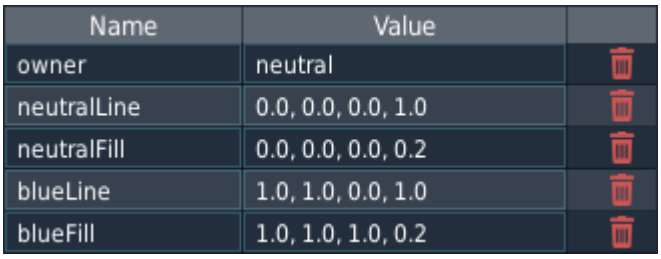
This is because the messenger triggers on change on changes in the value of the 'blues' flag. When the mission starts up, the messenger module initializes all messengers and for that it reads the current value of the input flag's values to detect changes. At this point, the value of the 'blues' flag is zero.

When later the ownedZones module starts up, it counts the number of zones owned by blue, arriving at 4. It then writes that value to the flag named 'blues'. A second later, the messenger inspects its input flag's value, and finds that it has changed to 4, triggering a messaging cycle.

Challenge:
Find a way to suppress the initial message.

The colorful zone

The final zone that is captured is initially colored black (outline 100% and fill 20%), and when it is conquered by the blue faction, the colors turn to 100% yellow (outline) and 20% white (fill). This is controlled by the zone's color override.



Note that the color definitions are DCS-style RGBA with values ranging from 0.0 to 1.0 (instead of the more common 0-255 or 0%-100% or even #FF000080 formats). The last value is for transparency, with 0.0 being completely transparent, 0.5 being half transparent, and 1.0 being completely opaque.

Challenges:

- Try to predict what color this zone assumes when red captures it
- Make the zone turn green 100% (outline), magenta 20% (fill) when it is captured by the red faction.

8.64 Owned Zones and Factories.miz (legacy migration)

8.64.1 Backstory

The original “ownedZones” module contained the functionality of conquerable zones, plus the ability to produce defensive and offensive units. As of version 2 of ownedZones, the latter production functionality was split off into its own ‘factory’ module.

8.64.2 Demonstration Goals

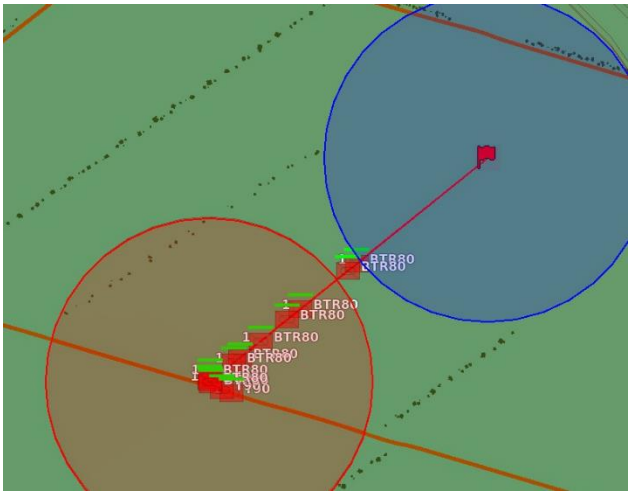
This module is a copy of an earlier demo “Owned Zones ME Integration”, simply updated for ownedZones 2.x and factoryZones 2.x and shows how you can easily and quickly migrate your own missions to the newer versions.

I recommend that you only migrate your mission if you want to take advantage of some of the new and improved abilities that ownedZones 2.x or factoryZone 2.x or later offer.

8.64.3 What To Explore

8.64.3.1 In Mission

To refresh your memory, you may want to run the demo mission “Owned Zones ME Integration” again to recap what it does, then run this mission to confirm that what they do is indeed identical:



When the mission starts, switch to F10 and what the events unfolding near Senaki: south of the airfield are two blue circles, the zones owned by Blue. As you watch, two red T-90 capture the southern blue zone.

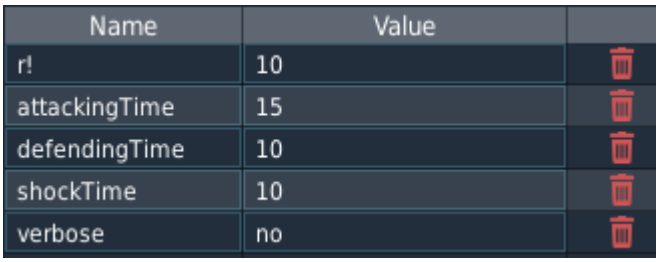
A short while later, one red infantry appears in the newly captured zone, followed by a string of BTR-80s that start to move towards the northern blue-owned zone. Eventually, that zone is captured. From that moment on, no more BTR-80 are produced in the southern zone.

So yes, this demo behaves exactly like the ‘original’.

8.64.3.2 ME

So how did we migrate from ownedZones legacy to split ownedZones and factoryZones? The way to do this is simple and straightforward:

8.64.3.2.1 Step 1: Migrating the config zone (when it exists)



- If there is a config zone for ownedZones, create a copy, and rename it “factoryZoneConfig”. This essentially copies all the attributes from the old (legacy) ownedZone which also contained the settings for the owned zone and the production attributes into a new config zone for factory zones. Since both modules retain full compatibility to their older (combined) parent, copying the config data over to the factoryZone config transfers all relevant settings. Any ownedZone specific settings in a factoryZone config zone are simply ignored. The same applies to ownedZonesConfig: all attributes that have migrated to factoryZone are now ignored, so you can safely re-use the old config zone unchanged.
- If you want to go the extra mile (this is optional), delete those attributes from both config zones that are no longer required (i.e. ignored). Not doing so will incur no penalties, just remember that those attributes are now ignored.

8.64.3.2.2 Migrating ownedZones

Again, this is only required for some of the owned zones: those zones that produce units. If an owned zone doesn’t produce units, it does not need to be upgraded. So, for those ownedZones that produced units in the legacy version, do the following steps:



- Add a ‘*factory’* attribute, no value.

That is all that is required, because similar to the config zone, the presence of the ‘factory’ attribute tells the factoryZone module to collect all information from the zone, and ‘stacks’ on the owned zone, using the same attributes that you previously fed to the legacy version of ownedZone.

So, adding a single attribute to a unit-producing legacy owned zone makes it compatible with the new ownedZone/factoryZone setup.

8.65 Caucasus Hangar.miz (StopGap)

8.65.1 Demonstration Goals

StopGap is a very useful tool to help you make your airfields look much more interesting – which can be an issue if you design your mission with many player slots: the airfield then usually looks empty. StopGap changes that, and this demo shows just that. Plus some of the niceties that the DML version offers over 'standalone'



8.65.2 What To Explore

Note: This demo expands on some of the concepts shown in 'No Gap, No Glory'

8.65.2.1 In Mission

Start the mission **in Single-Player mode**, choose Blue, and enter any aircraft that you like. Note that there are lots of aircraft to choose from (pretty much every module that DCS offers at the time of this writing).

Take a look outside and note the acute loneliness setting in: you are all alone on a nice airfield.

Now, chose communications→other→StopGap Contol→Enable StopGap.

Boom! You no longer are alone. The airfield is suddenly filled with a colorful collection of planes, and lots of them, all different (it may take a few moments for the aircraft to load their textures). If you remember the slot selection screen from a few moments before, you'll notice that all the aircraft that have appeared exactly match the ones that were on offer previously. That is not a coincidence.

Also, just for the record, note the absence of a Hornet on the taxiway, and three A-10A that are definitely not in sight in the forward row of aircraft on the western end of the tarmac. Also note that there is no Ka-50 helicopter sitting in the grass across the runway of the A-10A.

Choose communications→other→StopGap Control→Turn StopGap OFF.

And your are alone again, all the other aircraft have disappeared.

Now try the same with some other module that you own – you should be able to select it as a slot. Enter the aircraft – you are again on an empty airfield, in a different spot than before.

Enable StopGap again to populate the airfield. Spot another plane that you can enter (i.e. own), and change slot to that aircraft.

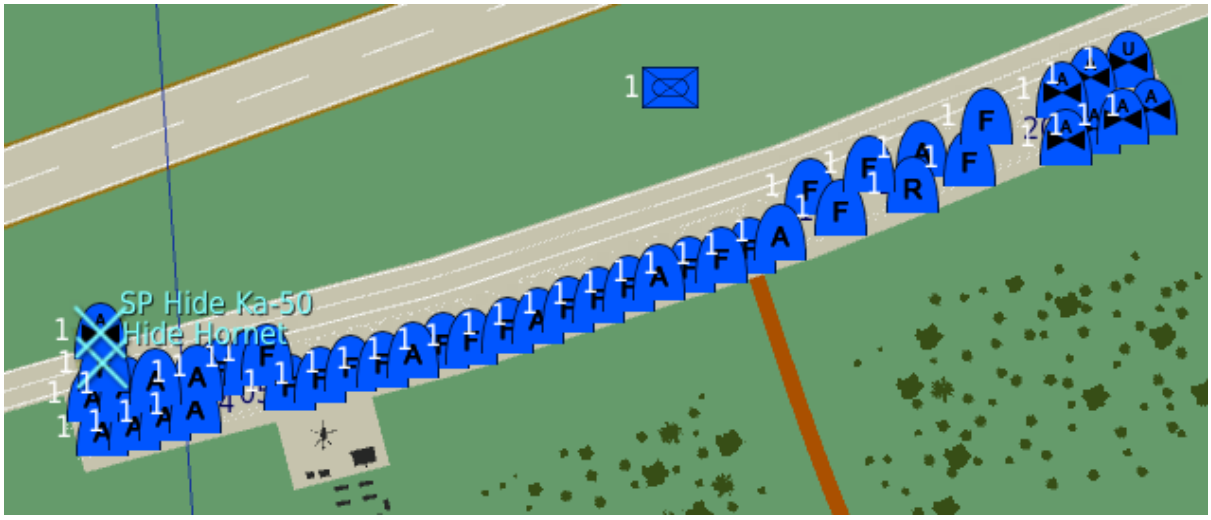
Note that you do not have to disable StopGap in order to change to another aircraft, and that the plane that you left is filled again with a stand-in. Should you have moved your plane before switching slots, the stand-in is returned to its original position (i.e., the one that you placed it in ME)

Now exit the mission, **and re-start the mission in multi-player** (you can do that from inside Mission editor) and **ensure that StopGapGUI** is running. Choose the A-10A again, and make all planes visible.

Note that although there is not Hornet on the runway, there now sits a Ka-50 in the grass across the A-10A.

8.65.2.2 ME

Just how significant a change stopGap introduces to your mission design you will realize when we look at the tarmac layout in ME:



All planes are player-controlled ‘client’ skill. This means that without stopGap, their position on the airfield must remain empty: you can’t place a static plane above the slot, or the player plane can’t spawn there. That is the reason why traditionally, multi-slot mission airfields look so empty. StopGap fills every client plane with its static look-alike until a player jumps into that slot.

8.65.3 Discussion

There are a couple of intentional exceptions. On the western part of the tarmac, in the front row, there are three A-10A, and in front of them, an F/A-18 Hornet. Neither of these has a static representation, even though you *can* choose to slot into them.

The reason for this is that they each use a method to selective disable stopGap for them:

- Hog: by using "-sg" as part of their unit name
- Hogs: by using "-sg" as part of their group name
- Hornet: By sitting inside a trigger zone that has the '*stopGap = false*' attribute. This (superior because obvious and more elegant by orders of magnitude) option is only available in the DML version of stopGap, while the mundane options are also available to 'standalone' stopGap



- Ka-50: Like the Hornet, the Black Shark sits inside a trigger zone. Unlike the Hornet, this trigger zone has an attribute stopGapSG = false. This option is also only available to DML, and allows you to selectively disable StopGap for all units inside this zone in single-player more only. This can be very useful for some maps where planes 'fall from the sky' in single-player mode, but work well in multiplayer due to StopGapGUI's superior synch feature. Note that this option can be used without Zones by adding "-sp" to the unit or group name – but that is not half as convenient as using zones.



There are a couple of caveats when you use stopGap:

- Place client aircraft as 'from ground hot/cold'. That way you can control the location where the player spawns, and the orientation (heading) of the stand-ion static. Never place a 'from ground' plane in a closed bunker: the player can't exit the bunker due to a bug in DCS.
- Use single-unit player groups. This is strictly speaking not necessary, but it helps to avoid side effects. StopGap supports multi-unit groups. Since stopGap only supports player aircraft, an AI aircraft will not be replaced with a static. Also, groups that contain AI aircraft cannot be placed as 'from ground', which will prevent you from correctly aligning the aircraft, and subject the entire group to DCS's vagaries of spawn locations (meaning: they will be unpredictable, spawn at some location at the airfield, often not the one that you want).
- When you place aircraft at FARPs, take care that your FARP is at a location with even ground. A bug in DCS can wreak havoc on planes that spawn on slopes.

Performance Considerations

Usually, the performance impact from StopGap is minimal – provided, of course, you don't overdo it. Since stopGap uses static aircraft instead of AI aircraft as stand-ins, the performance impact from the planes that you see is dictated more by the number of *different* plane models (DCS has to load geometry and textures for each) and available memory in the player's GPU than the absolute number of planes that are replaced by statics. Therefore, 20 F/A-18 on the airfield often hit a player's performance less than 10 aircraft that are all of a different type. In other words, this demo pretty much represents a worst-case scenario imaginable for a stopGap-augmented airfield: more than 40 aircraft, of which 40 are different

types. From how bad this demo impacts your machine you can gauge the impact on missions you are about to enhance with stopGap.

In any event it may be a good QoL feature to also allow players to turn stopGap off – just like this demo does. That way, a group with marginal computers can decide for themselves if they prefer performance over splendor.

8.66 Sitting Ducks in a Barrel.miz (SittingDucks)

8.66.1 Demonstration Goals

SittingDucks is an enhancement to missions, made possible by the stopGap add-on: it allows ‘destructible player slots’. As such, it *requires* stopGap, and only works in multiplayer. Server-side, it also requires the SSB plug-in to block the slots.



8.66.2 What To Explore

8.66.2.1 In Mission

Ensure that SSB and stopGapGUI are running on the server

Start the mission as multiplayer, and survey the possible player slots

BLUE COALITION | 0 players | PLAYERS POOL

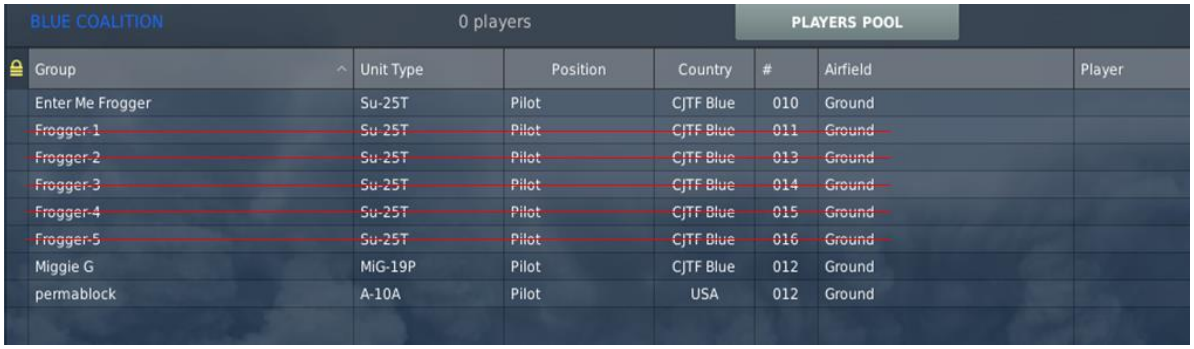
Group	Unit Type	Position	Country	#	Airfield	Player
Enter Me Frogger	Su-25T	Pilot	CJTF Blue	010	Ground	
Frogger-1	Su-25T	Pilot	CJTF Blue	011	Ground	
Frogger-2	Su-25T	Pilot	CJTF Blue	013	Ground	
Frogger-3	Su-25T	Pilot	CJTF Blue	014	Ground	
Frogger-4	Su-25T	Pilot	CJTF Blue	015	Ground	
Frogger-5	Su-25T	Pilot	CJTF Blue	016	Ground	
Miggie G	MiG-19P	Pilot	CJTF Blue	012	Ground	
permablock	A-10A	Pilot	USA	012	Ground	

Jump into “Enter Me Frogger”

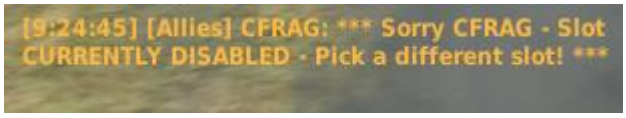
Look to your left. Notice the five Su-25T Frogfoots. These are the player slot stand-ins for player slots “Frogger-1” through “Frogger-5”

Look to your right. A single Mig 19P that represents the player slot “Miggie G”.

Choose Communication→Other→Blow Froggers on the Left. After the explosion, note that all Su-25T on your left side have gone – they were destroyed.



Within 30 seconds, try to enter one of the Su-25T (Frogger-1 to Frogger-5). The server denies you access:



Wait until the aircraft re-appear (may sometimes be difficult to see, as the still-raging but harmless flames may obscure them). This happens some 30 seconds after the explosion.

You should now be able to enter the cockpit. If you experience some strange lighting effects, these come for the flames. They do not affect the aircraft and will be removed soon.

If you have access to the Mig-19P module, you can try the same with the plane on your right (blow it up, try to slot in, wait until it re-appears, re-slot)

Notice that there is one player slot for an A-10A, “permablock” that you can’t enter, and that has no visual representation on the airfield.

8.66.2.2 ME
Tho only thing noteworthy here are:



- the permanently blocked Hog, which is accomplished by ‘manually’ setting the flag ‘*permablock’* (which matches the A-10A’s group name) to 100, making it inaccessible via SSB. This helps you to test if SSB is functioning correctly – you should not be able to be seated in that aircraft past second 1 of the mission. It also tests that stopGap runs correctly, as stopGap does not display blocked aircraft. If you can slot into, or see the A-10A on the airfield, something is amiss.
- In the sittingDucks’ config zone, we have set *resupplyTime = 30*, so any player slot is blocked for 30 seconds after destruction of the stand-in, and then becomes available again. By default, this value is -1, meaning that aircraft once blocked, stay blocked. Here we simulate a re-supply after an incredibly short time span of 30 seconds. In a normal mission, you’d set this value to something more sensible: 3600 (an hour) or more.



8.66.3 Discussion

SittingDucks enables new strategies in long multi-player missions. Due to the fact that it relies on SSB, it is multiplayer-only, and it also requires ‘stopGap’ to run to show the stand-in aircraft.

Once you have navigated those obstacles, your missions can gain a significant new aspect: attacking an enemy player air base becomes strategically attractive, and defending your own player slots (especially strategically high-value aircraft) becomes an important goal.

From an ME perspective, these modules are drop-in-and-forget.

8.67 Flag Score.miz (Player Score: coalition scoring via flags)

8.67.1 Demonstration Goals

PlayerScore not only keeps individual player's scores, but also a running total of points awarded to a coalition. Mission designers can directly add or remove points to and from a coalition by using flags. This demo shows you this feature in action.

8.67.2 What To Explore

This mission builds on the demo 'Later Score', so make sure that you have at least glanced at what that mission does to get a better picture.

8.67.2.1 In Mission

Run the mission and enter your favorite blue cockpit. Immediately notice the message that PlayerScore puts out during startup:



In the cockpit, go to Communications → Other → Add score to Faction → Score for Blue. A message appears:



Next, choose Communications → Other → Add score to Faction → Red Scores

No new message appears (be advised that in the background, and without notice to players of the blue coalition, score is added to Red)

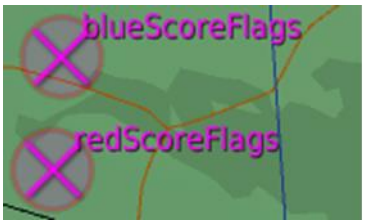
Now change faction to red, enter the Frogfoot's cockpit and again choose Communications → Other → Add score to Faction → Red Scores

This time, a message appears:



8.67.2.2 ME

The critical item here are the two trigger zones redScoreFlags and blueScoreFlags that each tell PlayerScore which flags to watch for a change in value, and then tell PlayerScore how many points to award when the relevant flag's value changes.



Name	Value
blueOne	500

Name	Value
redOne	2000

So in this mission, blue receives 500 points whenever the value of flag "blueOne" changes, and red receives 2000 points whenever the value of flag "redOne" changes.

Instead of assembling some complex conditions to set those flags, we simply use a radioMenu to generate flag changes for red and blue.

Note multiple changes to a scoring flag will add that flag's score amount multiple times to that faction. That is the reason why, after we already invoked "Red Scores" on the blue side (which changed the flag "redOne's" value but did not result in a message to blue), invoking it again on the red side changed the current score for red to 4000 (2000 for the first time when we were blue, and another 2000 when we invoked it as red)

8.67.3 Discussion

You may wonder what happens if you enter an illegal score value into the flag/score table for red or blue. It's easy to mistype, or accidentally enter a letter into the numbers. In those cases, the score that is to be added would be an illegal number, something PlayerScore doesn't know what to do with.

The demo has one such intentionally malformed score value in the blueScoreFlags table, the value for the "noNum" flag.



PlayerScore deals with this in two ways: First, it sets any score that it doesn't understand to zero, so triggering "noNum" will not stop your mission, merely award zero additional points to blue.

Second, it warns you at mission start that it has encountered a flag score value of zero – this is what you saw when you started the mission.



Challenge:
Create a flag "blueWins" that awards 10'000 points to blue, and at the same time reduces red's score by 100'000 points

8.68 Take on TACAN.miz (TACAN)

8.68.1 Demonstration Goals

Similar to the NDB module, DML can place a TACAN on-demand on the map, and even (should you find it useful) randomize the heck out of it (which can be counter-intuitive at times since – unless used to mark target locations – TACANs are placed at known locations).





This demo aims to first demonstrate the default use case of placing a TACAN nav aid at a fixed location, with a fixed channel and mode. It then proceeds to show how a TACAN can be randomized, and how the built-in TACAN GUI can be used to retrieve a TACANS callsign, channel and mode interactively.

8.68.2 What To Explore

8.68.2.1 In Mission

Note:
This demo requires a TACAN-capable aircraft. The free Su-25T that comes with DCS does not support TACAN radio navigation, hence if you sit inside the Frogfoot for this demo, you'll have to believe what I write, and can't follow along. The demo mission provides player slots for the Hornet, Hog, Viper, Mudhen and Tiger who are all TACAN capable.

Enter a TACAN-equipped module, or the Frogfoot. If you using the Frogfoot, some parts of this will be a thought experiment.

Set your aircraft up for TACAN navigation, and tune to channel 74X. If your equipment supports channel callsigns, you should see "BND", the needle point in an eastern direction, and your distance measuring equipment will show some 16nm.

Recommendation: Press "Active Pause" to safely perform the next steps without having to worry about crashing your aircraft.

Switch to F10 Map view. Inside the red quadrangle there should be no TACAN unit, and to the east, roughly between Senaki and Kutaisi there is one TACAN unit. You are currently tuned to that TACAN.

While in map view, choose Communication → Other → TACAN → Deploy TACAN. A new neutral (grey) TACAN unit appears somewhere inside the red enclosure, and a message similar to the following appears:



Note that the callsign can be any of the following: “TXB”, “AKA” or “NUI”, the channel any of 2, 3, 5, 6, 7, 8, 9 or 12. Mode is always X.

Tune to this TACAN to see that it works, and use the map’s metering tool to confirm that DME also works correctly.

Now choose Communication → Other → TACAN → Deploy TACAN again. The TACAN inside the red quadrangle disappears, and is replaced by a new one, at some other location, along with a new message similar to



(again, callsign and channel may vary inside the above mentioned parameters)

Now choose

Communications → Other → TACAN → Remove TACAN. A message similar to

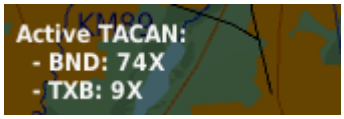


appears, mentioning the TACAN that you deployed last (“NUI” in this case”. Check on the F10 map that the TACAN unit has disappeared.

Now, deploy a new TACAN inside the red zone



Now choose Communications → Other → Available TACAN stations:

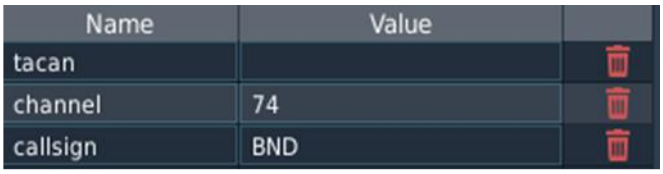


The second station listed will be the one that you deployed last, the callsign and channel will probably differ in your playthrough.

8.68.2.2 ME

We start with the default use case: the fixed TACAN placed with the “Bend TACAN” trigger zone:



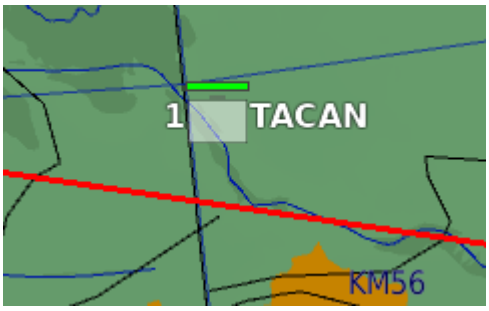


This TACAN is the minimal use case, and the one you will probably use most in your missions. It sets up a TACAN as follows:

- Position at the center of the trigger zone
- Channel 74
- (Mode X – default)
- Callsign “BND”
- (onStart is true – default)

This TACAN deploys on mission start, and is immediately added the available TACANs list.

Now let’s turn our attention to the ‘jumping’ TACAN, the one that is defined by the quadrangular “Test TACAN” trigger zone. As we have seen, it behaves markedly different from the above example. Here’s how it works:



- the `rndLoc` attribute causes the TACAN to spawn randomly inside the trigger zone. This demo uses a quad trigger zone to show that this, too is supported. The red frame is just for visualization purposes.
- The `channel` attribute now lists multiple possible channels, and even includes a range (“5-9”). Each time a new TACAN spawn, a new channel is picked randomly from this list.
- Similarly, the `callsign` attribute lists multiple callsigns, and like before, one is chosen by random
- Mode defaults to X (no `mode` attribute)

Name	Value
tacan	
rndLoc	yes
channel	2, 3, 5-9, 12
callsign	txb, aka, nui
deploy?	start
destroy?	stop
announcer	yes
onStart	no

- `deploy?` and `destroy?` are inputs that are triggered by a change on the flags that are listed here. If you look at the `radioMenu` module defined in the “Tacan Menu” trigger zone, you’ll find that they connect to `itemA` and `itemB` via the `start` and `stop` flags.
- Since announcer is true, each spawn of a TACAN station from this tacan zone will have the “NOTAM” announcement with callsign, channel and mode.
- Because `onStart` is false, the red quadrangle is empty after mission start, it must be populated with a signal on `deploy?`.

Which leaves us with the player interface that allows them to list all currently active TACANS that DML has placed. This is done with a simple global attribute entered in tacan’t config zone:

Name	Value
GUI	yes

- The GUI attribute, when set to true, installs a communication item for all players that lets them list all known (to their side) TACANS placed by DML.

8.68.3 Discussion

Randomizing a TACAN is an edge case outside of training missions, but it can make for nice 'TACAN Hunt' missions (where the player is tasked to find and destroy enemy TACANs before they can be used to vector in CAS missions). So here we have some related challenges:

Destroy a TACAN and have it noticed

Destroy a TACAN and have a messenger trigger when that happens. Hint: this will usually involve using the "trackWith:" attribute

Replacement Killers

This demo removes a previously spawned TACAN before it spawns a new one. Have new TACANS spawned without the previous one being removed.

Tricksy Basterds

Note that it is possible to have TACANS spawn in close proximity to each other that have conflicting channels and/or callsigns. This can cause grave inconveniences to pilots who aren't aware of this. Note also that DCS models radio transmission realistically. A TACAN signal can be blocked by mountains. Use both of these to create a mission where pilots need to find a TACAN station in a valley that have enemy misdirection transmitters deployed.

8.69 Civ Air International (Civ Air 2.0 “Off-map” locations)

8.69.1 Demonstration Goals

This demo highlights the new abilities Civ Air gained with version 2.0 – namely off-map ‘connectors’ that allow civilian flights to seemingly come from an off-map location or depart the map.



8.69.2 What To Explore

8.69.2.1 In Mission

Start the mission and enter the ‘Observer’ Su-25T. Then switch to F-10 map view and zoom out as far as you can. You’ll see at least 15 white (neutral) flights across the entire map, with new flights being added every 20 seconds until there are 30 neutral flights.

Click on the flights to see where they originated and where they are heading by inspecting their name. The name of each flight tells you where they took off from and what their destination is. You’ll note that in addition to flights between Caucasus airfields,

- flights can appear to come from locations that are not part of the Caucasus map (e.g., Zürich, Tokyo, Singapore, Stockholm)
  GROUP Zürch-Sochi-Adler/5

- Flights can be allegedly (by their name) headed to locations that are not on the Caucasus map (Tokyo, Zürich, etc.)
  GROUP Sochi-Adler-Tokyo/16

- There are flights that originate and are heading to off-map locations, i.e. will not land on the map
  GROUP Athens-Stockholm/13

If you observe a flight that is headed to an off-map location (e.g. Zürich), you’ll notice that when it approaches the map’s edge, it disappears-

Likewise, you’ll observe flights appearing at the edge of the map, and these flights will invariably claim to have originated from an off-map location.

8.69.2.2 ME
There are few surprises here: The flights that enter the map from or depart the map to off-map locations are created by the new 'inbound', 'outbound' and 'in/outbound' civAir attributes.



If you look closely, you'd see that I placed these trigger zones roughly in locations where flights that were coming from or heading to those locations would enter/leave the map: at the edge, somewhat aligned with the route that an aircraft that traverses the black see would take off-map. Also, on inspection you will realize that the name of the trigger zone is what is reflected in the flight's name.



When looking at the trigger zone's attributes, there is also little surprise. The civAir attribute makes it a civAir zone, and the in/outbound keyword tells civAir that flights can pop into existence inside this trigger zone, and disappear when they are heading for this zone and are inside of 5km of the zone's center. I've used in/outbound for each 'outside connector' in this map, so each of those trigger zones can serve as a source for aircraft as well as a destination.

8.69.3 Discussion
There are some interesting points to look at:

It's just an illusion:
I have intentionally placed the in/out' zones close to the map's borders. That way, flights seem to leave and enter the map – they create that illusion. That does not mean that you have to place them there, not that flights materialize or disappear only at the map's border. If you want to, you can place those trigger zones anywhere on the map, and planes will pop in and out of existence just like you tell them. I've seen little use in doing so, but your mission may use this for a cool trick.

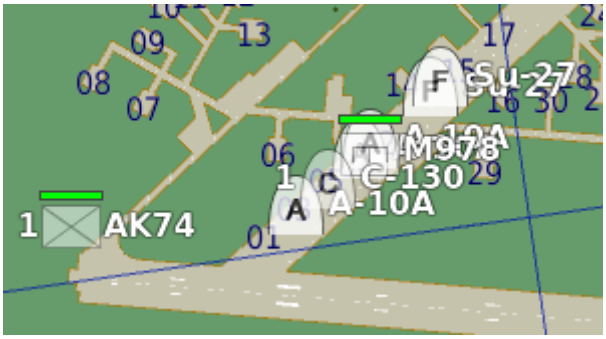
Say what country?
Civ air picks the nationality of the civilian flights from your mission's list of neutral factions. That can mean that, if you do not curtail the list of neutral countries accordingly, some unexpected countries may pop up – for example the Third Reich. To avoid that, simply move unwanted countries to either red or blue so that Civ Air cannot pick them as host for the flight.

No touchie touchie!
A setup that has both inbound and outbound locations can have flights that do not touch down on the map at all – they will traverse from one 'connector' to another. You can even make this intentionally if you provide no airfield to land or depart from, and then all civilian flights will simply traverse the map without ever touching down.

8.70 Airbase mine.miz (airfield)

8.70.1 Demonstration Goals

This mission demonstrates how the airfield module can generate signals that other modules can use to trigger actions when airfields are captured, how it can be used to assign ownership to a faction, and how it can be used to prevent airfields from being captured.



8.70.2 What To Explore

Note: parts of this mission demonstrate the airfield module's ability to persist data. If you want to follow all examples, your DCS installation must be de-sanitized. If you do not de-sanitize your DCS installation, the mission will still run, but saving (persisting) and restoring will not work as described.

8.70.2.1 In Mission

Enter your trusty Frogfoot (located in Kutaisi). Switch to F10 Map view, zoom out and pan over to Senaki Kolkhi. Notice the neutral units on the airfield and observe the lone blue Marder type IFV approaching Senaki Kolkhi.

When that Marder gets sufficiently close to Senaki Kolkhi, the mission's ground capture constraints are fulfilled, and Senaki Kolkhi is captured by BLUE. A message comes up:



Looking at the airfield you notice now that the units that formerly were neutral now all have been replaced by blue units of the same type. That Hercules has reset and is now preparing to depart Senaki Kolkhi

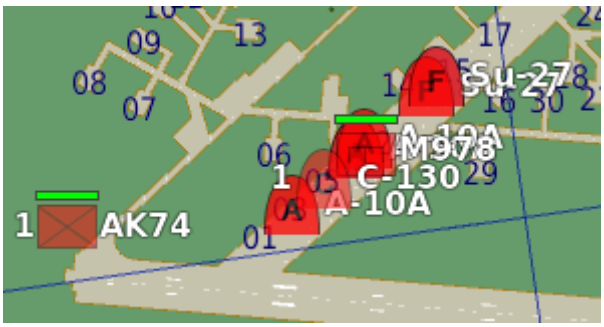


Now let's take matters in our own hand. Choose Communications → Other → Set Senaki Ownership → Give Senaki To Red.

A message comes up



And a look at Senaki confirms now that all units at Senaki are red (note for clarity: the units did *not* change ownership, they were deleted and re-spawned as a different coalition; in current DCS, units cannot change their affiliation).



If you let time proceed, the Marder IFV will continue it's approach to Senaki, and start fighting with the red AK-74 soldier – a short fight.

Note that now, although BLUE now fulfills ground capture rules (the IFV is the only ground-based fighting vehicle in the airfield's 2km radius), BLUE does not capture the airfield, it stays in RED's possession.

So let's relinquish RED's grip on Senaki-Kolkhi. Make sure that the Marder has reached its destination (no longer moves) and that it has destroyed the Ak-74 unit.

Choose Communications → Other → Set Senaki Ownership → Whoever can capture

A few second later this familiar message comes up



More importantly, the units at the airfield now are again blue. The AK-74 has respawned in blue.

Now cast your eyes on Batumi in the south. Click on the airfield to confirm that it is neutral. Note the BLUE Marder that is well within capturing distance, yet Batumi is still neutral.

(The remainder of this demo works only if you have de-sanitized DCS)

Now choose Communications → Other → Set Senaki Ownership → Save Mission Status.

Quit the mission

Start the mission anew, enter the Frogfoot, zoom out, and inspect Senaki-Kolkhi.

Note that Senaki Kolkhi is owned by blue (not Neutral), just as it was when you saved the mission.

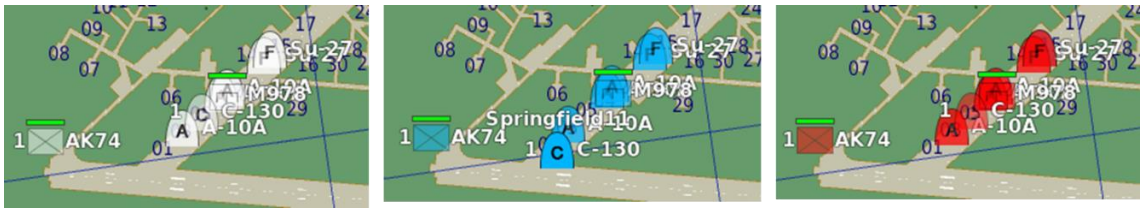
Inspect Batumi, and confirm that yes, it too is still neutral even though the blue Marder is well within capture distance.

Finally, for the mission to start from the beginning, exit the mission, then remove the folder “demo - airbase mine (data)” from your “missions” directory

8.70.2.2 ME

The Changing Affiliation Effect

In Senaki Kolkhi we can observe an interesting effect where units appear to change their affiliation when the airfield changes ownership: from Neutral to Blue to Red.



Since units can’t dynamically change their coalition, this is clearly the result of some creatively (and purposefully) arranged modules. This is how we can make it happen:

Initially, Senaki-Kolkhi is populated by neutral forces. This is done with a cloner that is configured this way:

Name	Value
cloner	
clone?	senakiOwner
preWipe	yes
masterOwner	Senaki Owner Control
onStart	yes

- The cloner *does* perform a clone cycle at startup, as indicated by onStart = yes. This produces the neutral statics and units.
- The cloner starts a new clone cycle every time that it detects a change in the senakiOwner flag’s value. This is one of the key attributes to achieve the ‘change population’s affiliation’ effect.
- Pre-wipe is enabled which removes all clones spawned in the previous clone cycle
- masterOwner is set to the trigger zone “Senaki Owner Control” which causes all spawned clones from this cloner belong to the same coalition who owns the trigger zone “Senaki Owner Control”. This is the key attribute for the ‘population affiliation’ effect for the cloner.

The second part is the airfield zone “Senaki Owner Control” that attaches itself to Senaki-Kolkhi (the airfield).

Name	Value
airfield	
makeRed?	makeRed
makeBlue?	makeBlue
ownedBy#	senakiOwner
autoCap?	anyone

- The airfield zone automatically associates itself with the closest airfield, Senaki-Kolkhi in this case.
- Invisible to us here, it also sets the zone’s owner (remember that all DML zones have an owner) to whoever currently owns the airfield – neutral in our case at mission start.
- We connect three flags to the inputs `makeRed?`, `makeBlue?` and `autoCap?` so that we can control these aspects from within the game

- We also constantly output the current owner of Senaki-Kolkhi to the flag ‘senakiOwner’ so that information is easily available to us.

Part of the magic happens when the cloner undergoes a clone cycle:

- It takes the template (the various units arranged inside its trigger zone) and prepares them for cloning.
- Next, since it has a `masterOwner` attribute, it fetches the faction that all cloned units should belong to from the zone named “Senaki Owner Control”, which is our airfield zone. The cloner then sets the spawning faction to a country of that faction, and spawns them.
- Since preWipe in the spawner is set to true, the spawner despawns any remaining units/objects from the previous clone cycle.

So the cloner always spawns units that belong to the same faction as the Senaki-Kolkhi airfield.

Now, when does the cloner undergo a clone cycle?

- Since `onStart = true`, there is an unconditional clone cycle at mission start. Here, the neutral units are spawned because at mission start, Senaki-Kolkhi belongs to the Neutral faction.
- `clone? = senakiOwner` connects the trigger input to start the next clone cycle to the flag “senakiOwner”. Since the cloner defaults to “change” as trigger method, any change of value in the flag “senakiOwner” triggers a clone cycle.
- “senakiOwner” is controlled by the airfield output ownedBy#, a ‘live output’ that always copies the current owning faction number (0 = neutral, 1 = red, 2 = blue) to the attached flag. The trick here is that since the cloner triggers on *change*, the live output is ignored when it’s the same value as before, but whenever the value changes to a different calue, the cloner triggers a clone cycle.

So when blue captures neutral Senaki-Kolkhi,

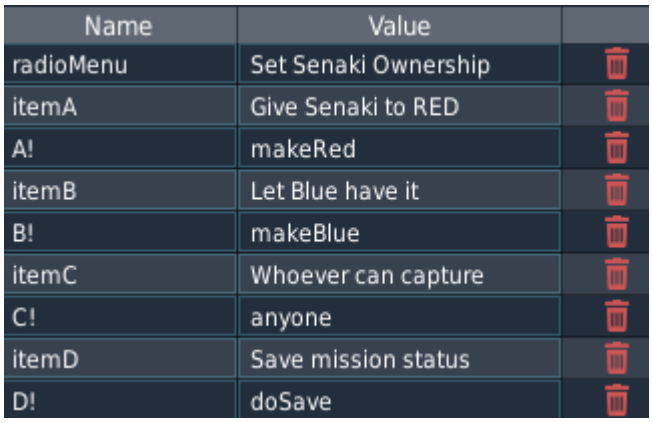
- the value of the flag “senakiOwner” changes from 0 (neutral) to 2 (blue), and
- this triggers a new clone cycle, with the following actions:
    - Since preWipe is true, all previously spawned clones are de-spawned
    - New clones are prepared from the template
    - The faction for the clones is determined from the ownership of the trigger zone “Senaki Owner Control” – the airfield zone that assumes the same coalition as the associated airfield: Senaki Kolkhi, which belongs to BLUE
    - All clones spawn belonging to BLUE

The result is that whenever the cloner undergoes a clone cycle, all units at Senaki Kolkhi seem to change ownership, and the cloner undergoes a new clone cycle whenever Senaki-Kolkhi is captured by another faction.

Assigning Ownership at runtime
In the mission we can give Senaki-Kolkhi to RED or BLUE simply by choosing the appropriate menu item from Communications.

This is accomplished quite easily with a radioMenu that connects to the inputs to airfield zone “Senaki Owner Control” to trigger the relevant actions:

- The output A! connects to the flag named “makeRed” which feeds into airfield’s `makeRed?` input
- The output B! connects to the flag named “makeBlue” which is connected to input `makeBlue?` of the airfield zone
- The output C! feeds the flag named “anyone” which is connected to the airfield input `autoCap?`
- Also, output D! feeds the flag named “doSave” which we have not yet looked at.

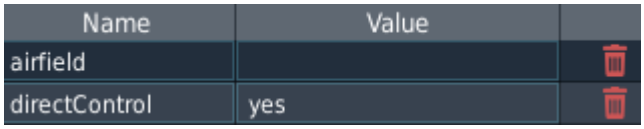


The radioMenu’s outputs A through C simply control who owns the airfield that is associated with airfield zone “Senaki Owner Control”. Since in our mission, the closest airfield this is Senaki-Kolkhi, we control ownership of the airfield Senaki Kolkhi. There are some important facts to remember: Should you set (force) ownership of an airfield to RED or BLUE,

- these airfields remain owned by that faction until you either force ownership to a different faction, or
- until you re-enable mission ground-capture rules by activating autoCap. That is why we have the third option “Whoever can capture” in our radioMenu

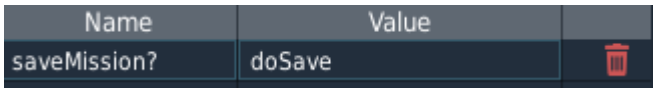
Batumi won’t budge

The airfield module can do more than assign ownership of an airfield – it can also make airfields unconquerable until you deem it convenient to make it capturable or assign it to another faction.



The trigger zone that accomplishes this for us is “Batuimi never budges”. All we are doing here is to set the `directControl` attribute to true. This disables mission ground capture rules and handles ownership control to the airfield module. This is the reason why the blue Marder that approaches Batumi does not capture the airfield even though it meets all ground capture requirements.

Persistence Support

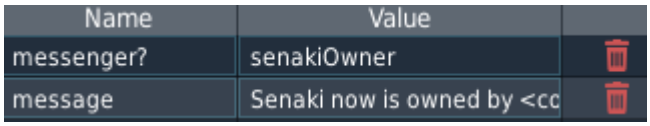


Airfield supports persistence out-of-the-box. There are no dedicated persistence controls in the airfield module. If you have enabled persistence in your mission and DCS installation (i.e, de-sanitized DCS as described in the ‘Persistence’ section of this document) and trigger saving, all airfield zones automatically preserve their state. That is what the D! output that connects to the ‘doSave’ flag accomplishes in the radioMenu above that is connected to the persistence module in persistenceConfig.

The Capture Messages (“Senaki no is owned by XXX!”)
Almost a side show is the message that appears whenever Senaki-Kolkhi is captured by another faction:



This is accomplished by a messenger module:



- messenger is triggered by a change in ownership, looking at the flag named ‘senakiOwner’ – this is the same flag that also triggers the cloner, and uses the same technique: trigger on change, and use the airfield’s `ownedBy#` direct output
- The message “`Senaki now is owned by <coa: senakiOwner>!`” itself uses the `<coa: >` wildcard to translate the value of flag “senakiOwner” to a coalition (“NEUTRAL”, “RED” or “BLUE”)

8.70.3 Discussion
The airfield module is merely the catalyst/enabler for most of the functionality that we see here:

- It prevents BLUE from capturing Batumi. We achieved this by associating an airfield zone to Batumi and setting its `directControl` attribute to true, which disengages the mission’s default capture behavior.
- The ‘units change affiliation’ effect is mostly an illusion effect where a cloner simply deletes all units spawned in the previous cycle and replaces them with new units form a different faction.
- We utilize the airfield module’s direct output `ownedBy#` to trigger the messenger module and cloner.

Remember that NEUTRAL can’t capture airfields and never fight to control them- they’ll cede to the first RED or BLUE ground vehicle. That is why we lock in Batumi’s faction with an airfield zone that turns on `directControl`.

Challenge:

- make Batumi capturable on command (use a radioMenu for this)

8.71 My Immortal (StopGap 'refresh' option)

8.71.1 Demonstration Goals

StopGap has an option to regularly 'refresh' all static slot replacement static objects with new ones, erasing potential damage that happened over time to those unoccupied player planes.



This option is helpful primarily for long-time server missions where such damage can accumulate over time, and is a purely cosmetic feature. This demo shows how this feature works.

8.71.2 What To Explore

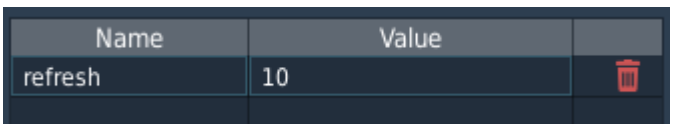
8.71.2.1 In Mission

This demo shows only a single feature of the 'StopGap' module, Please see the "No Gap, No Glory" and "Caucasus Hangar" missions for deeper insights into StopGap.

Note:
This demo requires that you own the UH-1 "Hewey" module.

- Enter the Huey, and switch to the left side gunner
- After the door opens, note the two planes close to you: a Hog, and a Falcon.
- Activate the mini gun and damage the Hog
- Note that after a few seconds, all damage to the Hog disappears
- Try the same with the Falcon. Note that you can damage the Falcon, but that the damage remains.

8.71.2.2 ME



The only relevant point is the added attribute '*refresh'* to StopGap's config zone. We have set it to 10, telling StopGap to refresh all static objects every 10 seconds. In your mission, you would prefer a setting of 3600 (once per hour) or

more, because this purely cosmetic effect can cost some performance when all static planes are re-drawn.

If you are designing a single-player mission or your server mission is designed for only a few hours, you can leave this option off (as it is disabled by default).

8.71.3 Discussion

Only Player Stand-ins are refreshed
As this demo shows, the refresh interval only ‘renews’ player slot stand-ins. The static Falcon in the demo is not refreshed, because it is not a stand-in but a static object placed in ME.

No sitting Ducks
Do not use the refresh option with Sitting Duck: the goals of the that module (make player slots vulnerable) and “refresh” work against each other, making results unpredictable.

8.72 Send in the Clones (requestable, airfield)

8.72.1 Demonstration Goals

This demonstrates the subtle differences between using requestable spawn zones and requestable cloners to create units that can be transported by Helo Troops. Additionally, it shows how the 'airfield' module nicely fits into the flow when we use cloners as requestable unit sources and keep an airfield's ownership locked to one side.



8.72.2 What To Explore

8.72.2.1 In Mission

NOTE:
To run, this mission requires that you own the UH-1 module

Run the mission, then

- Choose RED side, then
- Enter Red Huey Senaki
- Go to Communication → other → Airlift Troops and note that there are no troops to load into the Huey, nor are there any troops to request to spawn
- Change Slots on the red side to Red Huey Kutaisi
- Switch to map view, click on Kutaisi to bring up the airfield chart. Note that Kutaisi's owner is RED
- Go to Communication → other → Airlift Troops and choose "Request Kutaisi Cloners".
- A group of three red infantry spawns to the left of the Huey, and you can now load them into your chopper normally (via Communications)
- Once you load them aboard, a message "Clone Warriors are on board!" appears
- Switch to blue coalition
- Enter Blue Huey Senaki
- Enter F-10 Map view, and click on Senaki Airfield to bring up the airfield chart. Note that Senaki's owner is NEUTRAL

- Go to Communication → other → Airlift Troops and note that you can request units from the Spawner, but not from a cloner in Senaki.
- Request units from the Spawner. New units appear in front of you
- Switch to F10-Map View and note that formerly neutral Senaki is quickly captured by the newly spawned units
- Briefly (for at least one second) lift off, then set down the Huey again
- When you now go to Communication → other → Airlift Troops, note that now the 'Senaki Cloners' have become available as well

8.72.2.2 ME

Since this demo merely expands on everything we've already learned from the basic demos for Helo Troops, Spawners and Cloners, Let's merely go through the interesting bits:

No Red Requestables at Senaki

When you start the mission in the Red Huey in Senaki, there are no Troops you can request to be spawned. This is because Senaki is initially NEUTRAL, and the local **spawner** is keyed to the country ID 2, which is USA, and USA – in this mission – belongs to BLUE. Also, the two **cloners'** ownership is keyed to the airfield zone 'Senaki Owner'. Since that zone always has the same ownership as the airfield it is associated with, and Senaki (the associated airfield) currently is NEUTRAL, the requestable Cloners all belong to NEUTRAL, and thus are not eligible to a RED player.

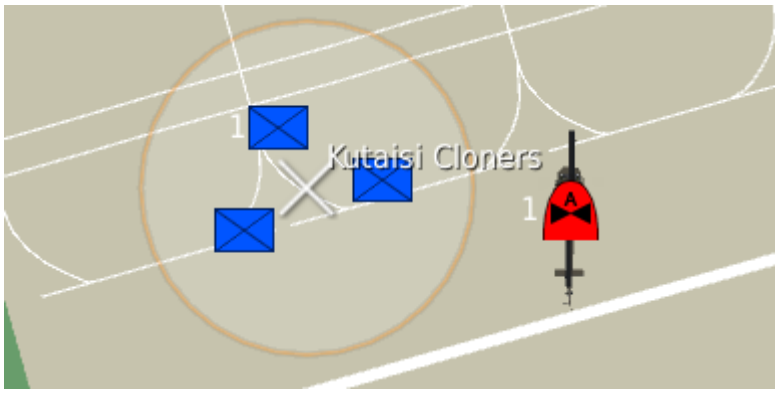
Name	Value
spawner	Soldier M4,Soldier M249,Sold
country	2
cooldown	120
requestable	yes

Name	Value
cloner	
requestable	yes
cooldown	20
masterOwner	Senaki Owner

That explains why neither cloners nor spawner are available to RED player in Senaki when the mission begins.

Red Kutaisi, blue originals

The cloner at Kutaisi is set up with three BLUE units, yet we expect to use it with a RED helicopter. I show this somewhat artificial setup simply to demonstrate that it is possible (units created by a cloner align with the faction that owns the cloner) to do something like this. On the other hand, due to the way that mission capture works, placing blue units on a red airfield can wreak havoc with your mission: during start-up these units exist for an instant while the cloner gets initialized and then removes the units.



Name	Value
airfield	
fixed	red

As a result, Kutaisi may randomly be assigned to blue – something that we do not want to happen. To prevent this, we also set an airfield zone that forces the issue by assigning Kutaisi to red (using the fixed attribute), and have the Kutaisi cloner follow the ownership of the airfield zone.

Name	Value
cloner	
requestable	yes
cooldown	20
masterOwner	Kutaisi Owner (fixed)
empty!	pickedUp

This creates a nice cascade: When the game starts, the airfield zone sets Kutaisi's ownership to RED and locks ownership down so that the blue units that may exist for a short while do not cause ownership issues. Kutaisi is owned by RED, and this is reflected back into in the Kutaisi airfield zone's ownership. The then cloner draws its ownership from the airfield zone, making it available to RED helicopter requests. Which in turn causes the units created when the player requests them also to be aligned with RED.

Initially no blue *Cloners*

Name	Value
spawner	Soldier M4,Soldier M249,Sold
country	2
cooldown	120
requestable	yes

When we first look at Senaki's requestable spawns from the BLUE helicopter, we find that the spawner is available to requests, but neither cloner is available. This is because the Spawner is hard-wired to country ID 2 (USA), and that `country` is a member of the BLUE faction, so blue players can access that Spawner even if it resides inside neutral territory (Senaki is NEUTRAL at the start of the mission).

The cloners, however, link their ownership to the airfield zone (masterOwner = Senaki Owner), which is associated with Senaki, and when the mission starts, Senaki is NEUTRAL. Hence, BLUE players have no access to these requestable cloners.

Senaki Ownership Status – why we need "airfield" zones

Well, it's complicated… From mission design standpoint, I want Senaki to be NEUTRAL when the mission starts up. The problem: much like in Kutaisi, where we place blue units to supply a template to the cloner, yet want Kutaisi be RED. In Kutaisi we use an airfield zone with a "`fixed = red`" setup to force the airfield RED, and make it remain RED (it cannot be captured by blue).

Name	Value
airfield	
fixed	neutral
autoCap?	releaseSenaki
raiseFlag!	releaseSenaki
afterTime	3

In Senaki we have prepared a couple of cloners with blue units, yet want Senaki to start as NEUTRAL. There is a residual chance (especially in the Multithreaded version of DCS) that due to the presence of blue units in Senaki at mission start (even though the cloner removes them at mission start), Senaki is handed to to BLUE.To prevent this, we also use an airfield zone, and use the '`fixed = neutral`'. But this poses a problem, since we want blue to be able

to capture Senaki. The solution is that we use a flag “releaseSenaki” that is raised 3 seconds after mission start, which turns Senaki back over to ground capture rules. This turns Senaki NEUTRAL at mission start and for the first three seconds of the mission, and after that time interval, Senaki becomes capturable – which happens as soon as a blue player requests units from the spawner (the cloners are inaccessible when Senaki is neutral, see above)

The need to lift off

After we deploy blue units from the blue spawner for the first time, Senaki is captured within a second or two, and theoretically, blue players landed at Senaki should be able to request units from the cloners as well. However, the cloners do not appear in the helicopter’s radio menu until the lift off and land again. The reason is purely technical: for performance, HeloTroops assembles a helicopter’s radio menu only when they touch down. So, when ownership of an airfield changes, the radio menu that should give access to the newly accessible cloners is only updated when the helicopter lands anew.

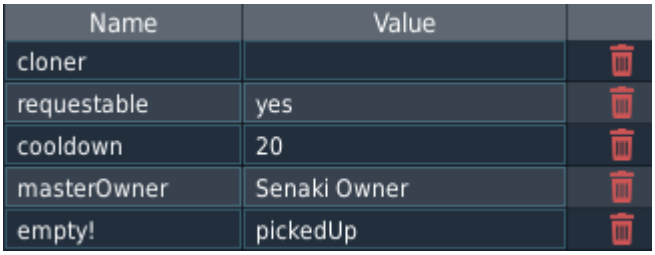
The no-show cloner

Although we have two requestable cloners at Senaki, only one of them is accessible to the Huey. The reason for that is that the units that are cloned by the cloner “Senaki Unorderable” includes an APC AAV-7 -- a unit that is not a member of the types that are declared as Helo Troops compatible (see ‘`legalToops`’ attribute for HeloTroops). Cloners are smart enough to suppress cloners that produce units types that cannot be carried by the helicopter in question, and the UH-1 can’t carry APC.

“Clone Warriors are on board!”



After you order troops aboard your helicopter that were created by a requestable spawner, a message comes up stating that the “Clone Warriors are on board”. This is produced by a messenger that simply waits for a signal on the flag “pickedUp”.



So who creates that signal? The requestable cloners, with their empty! output: when HeloTroops puts units inside a helicopter, it also tells a cloner that it has removed the cloned units, which causes the cloner to fire on the empty! output. This mechanism allows us to reliably verify that troops where picked up from a cloner by HeloTroos and allows for much better integration.

8.72.3 Discussion

Both spawners and cloners allow mission designers to provide interactive methods to soawn units that can be picked up by helicopters, and you might wonder what the differences are and when you should use a spawner over a cloner or vice versa.

Here's what differentiates one from the other

What	Spawner	Cloner
Unit Ownership	A spawner usually spawns units that belong to the coalition that the country belongs to that is specified in the spawner's 'country' attribute When using 'masterOwner' those units will belong to the same faction as the faction that owns the masterZone	Clones units usually retain the same coalition as the one that was given in their template. If there is a masterOwner given, all cloned units will belong to the same faction that owns the masterZone
Formation	All units spawn in the formation that is given in the 'formation' attribute	All clones appear as you arranged them inside the clone zone
Multiple Groups	A spawner always produces a single group	Cloners can produce multiple groups that even can belong to different factions. Be careful when you use this in conjunction with heloTroops, since it usually will give you headache
Orders	Spawners support the DML concept of Orders where you can iisue orders your spawned troops to defend, attack zones, or lase	Cloners do not support DML orders, their orders are whatever you give them in mission editor. Those orders usually do not mesh well with Helo Tropps
Picking Up Units with HeloTroops	Re-starts spawn cycle with cool-down	Produces a signal on empty! output

8.73 All is what I own.miz (ownAll)

8.73.1 Demonstration Goals

This mission demonstrates how you can use the ownAll module to gather owner information from multiple modules and assemble groups of owned zones to drive mission goals.



8.73.2 What To Explore

8.73.2.1 In Mission

Grab a cup of coffee, and make yourself comfortable. This mission purely consist of you watching things happen, and it will take a few minutes until everything is done.

- Enter the trusty Frogfoot at Senaki
- Switch to F-10 map view, zoom out, and that there are a total of five RED-owned zones, plus Senaki Kolkhi, which is denoted as Neutral by the white circle surrounding it.
- Note that I (for purely eye candy reasons) marked the area above the river as “North” and below as “South”.
- Go to Communications→Other→Starter, and choose “Start Blue Senaki”.
- A blue Hummer appears and approaches the ownership ring around Senaki. Soon, Senaki is captured by blue, and the following message comes up:
  Blue captured a NORTH base. Now holds 1 bases (0) of 3
- Go to Communications→Other→Starter, and choose “Start Left Blue”.
- Another blue unit appears that approaches the irregularly shaped red-owned zone south of senaki. Quickly, that zone is captured as well, an some messages appear, among them
  Blue captured a NORTH base. Now holds 2 bases (1) of 3
- Go to Communications→Other→Starter, and choose “Start Right Blue”.
- Again, a red unit appears, this time to capture the last remaining red-owned zone north of the river. Eventually, you’ll see two meesages appear:
  Blue captured all NORTH bases
  Blue captured a NORTH base. Now holds 3 bases (2) of 3
- Go to Communications→Other→Starter, and choose “Start Blue South”.

- Three separate groups of units appear, each one of them approaching one of the three red-owned zone south of the river: a FARP and two non-descript zones.
- A succession of messages appear, among them:
  Blue captured a SOUTH base. Now holds 1 bases in the South,
  Blue captured a SOUTH base. Now holds 2 bases in the South, and then in quick succession
  Blue captured a SOUTH base. Now holds 3 bases in the South,
  Blue captured all SOUTH bases and
  DONE!!!! Blue captured ALL bases

Check on the F-10 map that indeed, all red-owned zones to the north and south of the river (called 'Rioni', but the map unfortunately does not mention that) have been captured by blue.

8.73.2.2 ME
The object of this demo is that we understand how ownAll uses information provided by other zones (here owned zones, airfield zones and FARP zones – all zones that can manage their ownership) to establish who (faction) owns all and how many zones.

Additionally, we'll also look at the ability of ownAll to manage its own ownership, and use that ability to cascade ownership to other ownAll modules.

Divide and Conquer: ownAll's list-based approach
An important use case for ownAll zones is that they can be used to group (collect owned zones, or 'divide a map into smaller parts') a conglomerate of multiple owned zones, and can then give you a status with regards to who owns how many inside that group.

On the northern part of the map we have an ownAll zone "Own All North" that checks the ownership of the three zones "SenO", "Left" and Right".

Name	Value
ownAll	SenO, Left, Right
blue!	bNCapAll
blue#	blueNum
total#	baseNum

"Seno" is an airfield zone that always assumes the same ownership as the airfield it associates with, in our case Senaki Kolkhi (and this own is also the source for the circle of ownership that surrounds the Senaki-Kolkhi airfield).

"Left" and "Right" are Owned Zones that assume ownership according to the rules set forth in ownedZones. In our demo, they start as RED zones, and – since they are undefended - are conquered as soon as some blue ground forces enter the zone.

Name	Value
owner	red

So what happens when one of those three zones changes hands to blue? Each time that happens, the value of the flag that is connected to the "blue#" output is updated to the new number of zones that blue owns. Here, that new number is put into the flag named "blueNum". If we connect a messenger's "message?" input to that same flag, each new value is recognized as a change, and the messenger is triggered to display a new message.

In our demo mission, we have indeed set up a messenger to trigger on a change to the flag named “blueNum” – the messenger of the Zone “Cap reporter N”



So, each time that the value of the flag blueNum changes, this messenger is triggered (since we have no attribute to change it, this messenger defaults to triggering on “change”). the processed message for “`Blue captured a NORTH base. Now holds <v: blueNum> bases (<v:blueNumZ>) of <v:baseNum>`” is put on-screen for everyone to see.

Now, this message uses three wildcards, and we use them to illuminate some of the more subtle differences between ownAll and ‘plain vanilla’ owned zones.

The first time we see the message, is this:

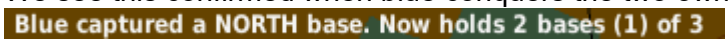


It appears that the message contradicts itself – “Now holds 1 bases (0)” doesn’t add up. The value “1” is returned from the flag named `blueNum`, while the value “0” is returned from the flag named `blueNumZ`. Where is blueNumZ set and what does it represent?



That flag is set by the ownedZone’s master process, as defined in the “ownedZonesConfig” config zone for ownedZones, and it is a direct value that always reflects the number of owned zones currently owned by BLUE. And this already nicely illustrates the difference between owned Zones and ownAll Zones: ownAll only counts the zones that you list in the ‘ownAll’ attribute, so you can build your own clusters – a much needed (and requested) ability vis-à-vis ownedZones that only offers this ability over the entire map (all ownd zones), and only counts zones of type ownedZones: the ‘0’ value that ownedZones outputs to the flag connected to “b#” direct output is (correctly) the number of owned zones that blue has captured: At this point in the mission we have captured Senaki Kolkhi – whose ownership is managed by “SenO” – an *airfield* zone, not an owned zone. ownAll does not make this distinction, and counts all zones that are listed in the ownAll attribute. The ownAllZone “Own All North” *does* list “SenO”, and accordingly, the output `blueNum` provides the correct number “1”

We see this confirmed when blue conquers the two owned zones “Left” and “Right”



and



At this point, blue has conquered all zones that are listed in the “ownAll” attribute of the zone “Own All North”, and that zone sends a signal to all flags that are connected to the “blue!” output, in our case the flag `bNCapAll`, which also connects to the messenger “Cap All N Reporter” and sends out the message.

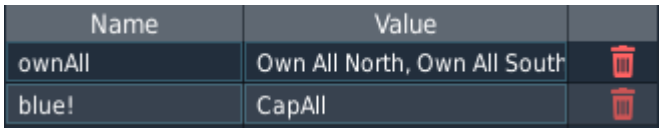


When we start the three blue groups in the south, we see something similar happening, except that this time, instead of an airfield zone, we have a FARP zone that manages the

ownership of the FARP, and whose ownership is monitored along with the two owned zones by the ownAll zone “Own All South”.

Cascading Fun

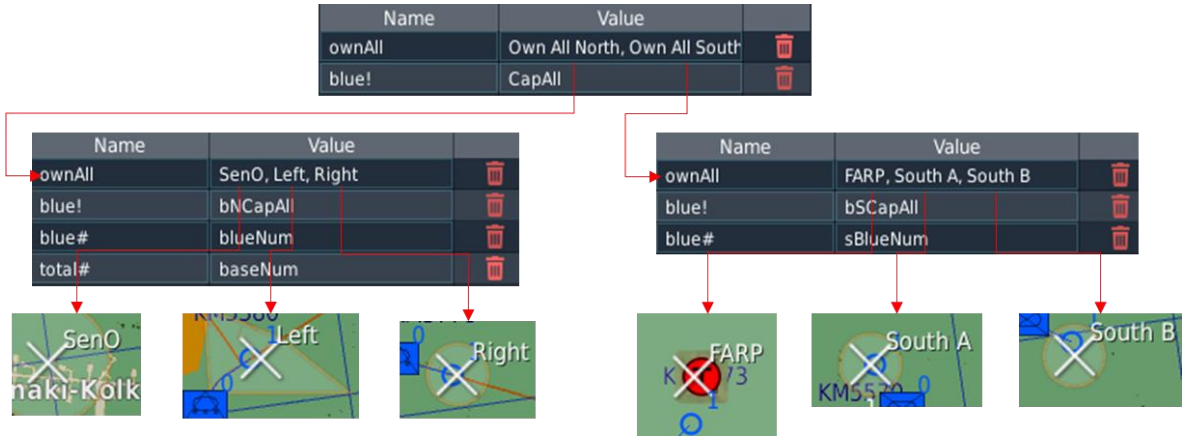
One important aspect of ownAll zones we haven’t touched yet: When one faction owns all the zones that are listed for the “ownAll” attribute, it not only sends a signal on the faction’s output, the entire *zone’s owner switches to the side that owns all the zones* listed. This may initially seem like a small design feature, but it becomes a tremendous help when you begin to assembly multiple ownAll zones into win conditions or similar decisions:



Let’s examine ownAll zone “Own Them All”. It lists only two zones: “All Own North” and “All Own South” – each ownAll zones themselves. Since each of these zones assume the ownership of the faction that owns all the zones that are listed in their ownAll attribute, we can easily use that “Own Them All” zone to signal us when blue owns all zones that are manages by the ‘subordinate’ ownAll zones “Own All North” and “Own All South”. In that case, “Own Them All” sends a signal on the output “blue!”, which connects to a messenger via the flag CapAll to output the message



In other words, we cascade ownAll zones to individually ‘account’ ownership of the Northern and Southern zones, and then use the ‘assume ownership of all-owner’ ability to communicate this upstream to “higher-level” own-all zones:

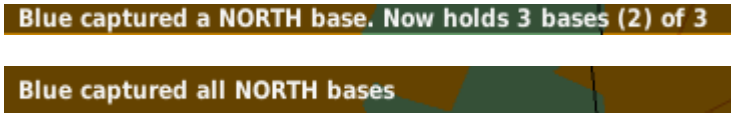


Of course you can then cascade to even higher levels. Use this feature if you divide your map into individual ‘ownership zones’ that can have individual goals.

8.73.3 Discussion

When we look at the messages that are being put out, we see a small annoyance, that we want (and can) polish out of the mission. Regard the following sequence of messages:





← REDUNDANT!

The third message “Blue captured a NORTH base. Now holds 3 bases (2) of 3” is redundant since it is immediately followed by “Blue captured all NORTH bases”.

So here’s a nice challenge for you

- Suppress the final “North captured…” and “Blue captured…” messages if all zones in that zones are captured, i.e. we only see the “Blue captured all North Bases” and “Blue Captured all South” bases, not the redundant third capture message.

- Now for a *real* challenge: Building upon the first challenge, also suppress the final (and only final) “Blue Captured all…” message when Blue has captured all bases.

8.74 Bombs Away.miz (bombRange)

8.74.1 Demonstration Goals

The bombRange module allows you to quickly add bomb range functionality to any mission, so this demo quickly demonstrates how you can quickly take advantage of that



8.74.2 What To Explore

8.74.2.1 In Mission

NOTE:
This mission also attempts to demonstrate bombRange's persistence abilities. In order to see those, you have to 'de-sanitize' DCS. If you decline to do so, you will simply not see this ability, but the mission itself will not complain.

Start the mission and jump into either the "Frogger" Su-25T (free) Frogfoot, or "Hog", an A-10A (license required). Your airframe is armed to the teeth with air-to-ground ordnance. Look ahead, and locate the two smoke markers, one white, the other green.

Close to either smoke are our target location that we are going to attack:

- White smoke denotes a 'bullseye' zone, where we try to place bombs as closely to the center as possible. Like an archery target, missing the center slightly will still award a hit, and unless we miss by too much, we receive a percentage.
- Green smoke denotes a single precision 'pinpoint' target that we must hit exactly, or fail, there are no percentages here.

In this mission I am flying the Hog, but you can just as easily use the Frogfoot. The only difference is that the statistics will be slightly different, as are the weapons names.



Fence in, go into air-to-ground mode and select bombs or rockets, then start an attack on the target area marked by the white smoke (the bulls-eye, try-to-hit-close-to-center type). As you approach, you will notice an armed watchtower – this marks the center of the target zone.

Adjust your aim accordingly (aim for the watchtower), and loose a pair of mk-82 or rockets.

Soon, your munitions will impact and you'll receive a note similar to

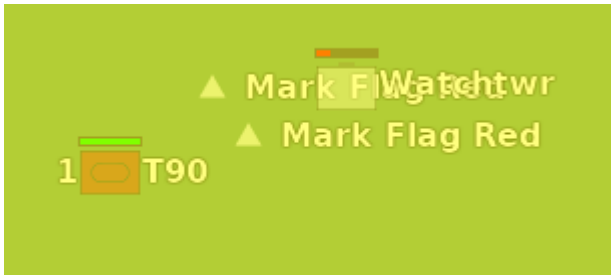
```
impact of Mk_82 released by New callsign from A-10A after traveling 0.47 km in 3.7 sec, impact velocity at impact is 129 m/s!
```

Which is quickly followed up by something similar to

```
INSIDE target area smoker, off-center by 17.2 m (Quality 65%)
```

Note that the numbers and zone names are likely to be different in your run. The main point here is that we receive a percentage, and some mildly interesting data that bombRange collects for you in case you are interested (when you cross-train aircraft, this data will become far more relevant for you).

Freeze the mission ("pause"), and switch to F-10 Map view. Zoom into the location that you have just attacked. It's easy to find, the target area is marked with a yellow circle. Zoom in until you can resolve both T-90 and Watchtower. You'll see something like this:



Note the two "Mark Flag Red" on the map. These are the impact points of your bombs. Return to the cockpit and unfreeze.

Come away, select your Mavericks (in the Frogfoot, activate and select your Vikhrs) and head towards the green smoke. Identify the tank, and loosen a missile (I assume that you know what you are doing and that you will hit it). A short while later, you will see something similar to

```
impact of AGM_65H released by New callsign from A-10A after traveling 5.27 km in 19.2 sec, impact velocity at impact is 238 m/s!
```

followed by

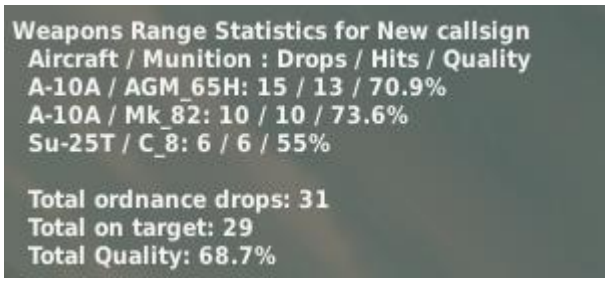
```
INSIDE target area Edgy V, hit on MBT Leopard-2A4
```

Continue attacking green and white smoke at your leisure, and make sure that you also use your dumb-fire rockets, maybe even switch airframes. Make sure that you continue flying for at least 30 seconds after your last hit.

Then end the mission.

Now re-start the mission, and enter one of the Ground-hogging air frames (“Groundhog” or “Groundfrog”).

Go to Communications→Other→Contact BOMB RANGE and choose “Get Statistics for <enter your callsign>”. You’ll see something similar to

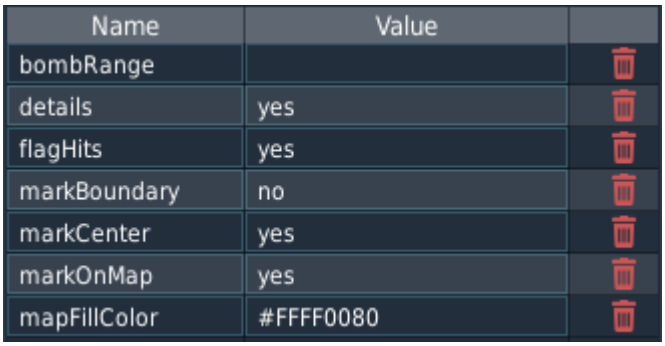


A nice summary of your exploits that you accomplished in your last mission – bomb range supports persistence. See also that bomb range remembers the airframe that you used for ordnance delivery.

Note:
If you have not de-sanitized DCS, persistence is disabled. The bomb range data from your previous mission is lost, and you will “NO DATA” instead

And that’s already all. We’ll

8.74.2.2 ME
Since the module does almost all the heavy lifting for you, the only really interesting things to examine are the two bomb zones. Let’s look at the ‘Bullseye’ zone first (the one that is marked with white smoke).



Note that I’ve added some attributes although they would not be strictly necessary; they are here to allow and invite you to do some experimenting. Let’s go through the list:

- details is set to true so we receive the highly detailed impact information
- flagHits is set to true so that we see the impact points on the map
- markBoundary is turned off (same as default, can be omitted)
- markCenter is turned on, and is responsible for the watchtower at the center of the zone.
- markOnMap is turned on, which in conjunction with mapFillColor is responsible for the yellow circle on the map (#FFFF0080 is a hex representation for yellow, 50% transparent in RGBA).

Note that the white smoke that marks this bomb range and the T-90 are not part of the target zone’s definition, I placed them manually.

This target zone is a ‘bullseye’-type zone: you are scored by how close to the center you can place the impacts in relation to the zone’s size – hence the name ‘percentage’ or ‘bullseye’ zones. Circular zones are always percentage/bullseye zones unless you change that with an attribute.

Note also that (should you not have managed to do that yourself yet) if you had placed a direct hit on the T-90, you’d scored a 100% hit even though that tank is not dead center in a percentage (bullseye) target zone. This is intentional so you can train object targeting and only receive a lower percentage if you miss the object.



Let us now contrast this setup with that from the “precision” quad zone. As mentioned in bombRange’s documentation, quad zones behave differently from circular zones in that they do not offer a percentage evaluation: they are inside/outside only. We couple this with the fact that target zones always report 100% inside hit when an object inside is hit and simply fence the object that we are interested in with the target zone. Now, when some ordnance strikes the object inside the zone, this is guaranteed to be reported as a hit – we have a surefire way to determine if the object was hit. Since it is a quad zone, it will always be either hit or miss – but the advantage of this arrangement helps us with one very particular issue that we have with precision bombing and high-velocity munitions that are often employed for this: they don’t really strike the ground, but at some (small) altitude above the ground. That’s usually not a problem for most free-falling ordnance, but high-velocity missiles tend to be not only very fast, but their trajectory tends to be shallow, and a projected ground impact will almost always fall outside of the target zone. Hence these precision zones that utilize objects inside them above ground to determine a hit.



By comparison, this target zone is almost anemic – and usually, you’d also forgo the ‘details’ attribute since most people are not interested in them. For enthusiast (and me while developing the module) the details are important, as the impact velocities for munitions can be very interesting (contrast the difference in velocities between a ‘Hydra’ dumbfire rocket and an mk-82 for an eye-opening experience.

A note on persistence
If your DCS is set up accordingly, you have seen that bombRange supports persistence: your past bomb results are preserved after you quit the mission. Since bombRange is a full DML citizen you get this ability for free when you add the persistence module, there is no additional setup required.

8.74.3 Discussion
A couple of observations before we get into some challenges:

In this mission, white and green smoke are classic ME smoke activated by triggers on mission start. I did this just to drive home how much more convenient and better DML smoke zones are. ME zones are ugly to set up, and expire after a mere five minutes.

Then I decided to keep disabled the boundaries for the bullseye zone (the one with the watchtower in the middle). I did that for two reasons:

- To better contrast that with the map view (yellow circle) and mainly because
- The markers that outline the circle make spotting impact marks difficult. So using both in the same mission is a design choice you'll have to make yourself every time.

Finally, although it is supported by BombRange, you did not have to check into the bomb range, it is always 'hot' and ready to score your drops. For single-player missions this is likely to be the best option, but you may want to require check-in for multiplayer and server-based (long-playing) missions.

Which brings me to our

Challenges

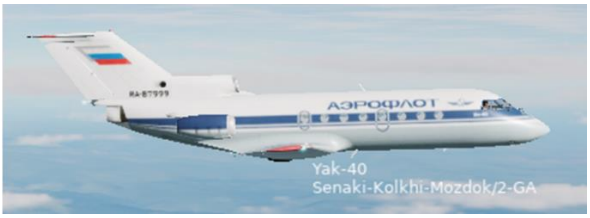
- Change the mission so that players need to check into the bomb range before their drops are scored and tabulated
- Turn on the boundary markers for the bullseye bomb range, and change them to windsocks. Then make sure that windsocks are also filtered and will not show up as (false) hits.
- Change the color of the map mark for the bullseye zone to orange, and make the outer rim disappear

8.75 Types and civil liveries (civAir liveries and type control)

8.75.1 Demonstration Goals

You can easily control which aircraft types should appear win which liveries in your mission. Here's a short example





8.75.2 What To Explore

8.75.2.1 In Mission

This is another 'sit back and watch' mission.

- Start the mission, enter your trusty Frogfoot, and wait for a couple of seconds to let civAir wait for the mission to settle
- Press F2 repeatedly to cycle through the various aircraft that civAir has spawned. Note that
    - They are all one of two types: a Yak-40 or C-130
    - The Yak-40 comes in two liveries: Aeroflot and Olympic Airways, while the Hercules is always dressed in Turkish Air Force colors.

8.75.2.2 ME

Usually, left to its own devices, civAir uses its default set of 'civil' aircraft types (few of which are truly civilian because DCS comes with a woefully small stock of civil aircraft, so civAir repurposes some military transports as civilian transports) and set of liveries for those aircraft.

In this mission, we see that not only the aircraft types have been restricted, but also their liveries.This is done in a two-step process:

First, we use a civil_liveries zone to redefine the available liveries for the Yak-40 and C-130. By adding attributes with their name to the civil_liveries zone, we replace their default livery entries (which contains the full set) with only the ones that we list here.

Name	Value	
Yak-40	Aeroflot, Olympic Airways	
C-130	Turkish Air Force	

If we left it just at that, the mission would still use all types of civil aircraft that civAir defines in its default set, which also includes the Herc and Yak-40. To suppress the default set and use only those types that we add in the civil_liveries trigger zone, we need to turn off the DCS default set in civAirs config zone.

Name	Value	
DCS	no	

And that’s already it. CivAir doesn’t load the DCS default set, and uses only the aircraft types that are listed as attributes in the civil_liveries trigger zone. And for those aircraft listed there, it replaces the liveries with those from the trigger zone. As a result, civAir uses only the aircraft types that you listed with the liveries you included for each type-

8.75.3 Discussion

Since CivAir is really simple to use, lets give you only an easy

Challenge:
Make the Yak-20 twice as likely to appear as the Herc, and add “Algeria GLAM” as a livery to the YAK-40

8.76 Boom Boom (explosion, map object destruction revisited)

8.76.1 Demonstration Goals

In the “Object Destruct Detection” demo, we looked at using the ODD module to trigger old-fashioned ME trigger rules to achieve a response. This time, we use the signal produced by an object destruct detector to properly (i.e. DML style) trigger a number of cloners to act when the map object (a building) was destroyed, and trigger a cascade of explosions to show off the groundExplosion capabilities..



8.76.2 What To Explore

8.76.2.1 In Mission

In “Object Destruct Detector” we connected an ODD module to an ME trigger to show how ODD can be used in old-fashioned “vanilla ME” style mission building.

One of the reasons was that then we didn’t really have a DML module to blow stuff up properly. With the advent of groundExplosion that has changed, and since I also upgraded ODD at the same time, it was time to put the two together in the same arena to show off some fabulous effects that can even rival some major productions.

Accordingly, your only role in this mission is to watch the action unfold. For the entirety of its uh, 25 seconds…

Run the mission. You’ll start the show by looking at a truck in front of a warehouse. In five seconds the office building forward and to the right of the truck is going to explode. So zoom out, and sweep the camera so that you can see more of the scenery to the right.

Boom! goes the building, and across the yard, to the left of the truck a series of smaller explosion rips through the oil tank storage. Explosions echo, and after a short while, you see three fire trucks moving towards the source.

Eventually, they arrive, and arrange themselves artistically to make the scene look good.

Run the mission again, and now observe a couple of important details:

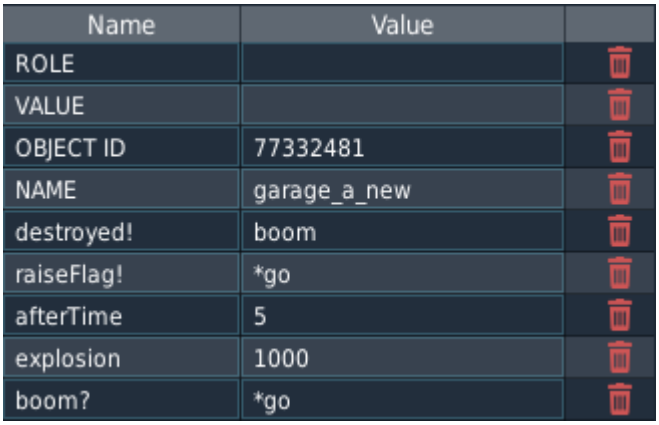
- The fire trucks to not appear until the office building is destroyed

- The series of explosions that rip across the yard where the oil tanks are located is randomized
- If you turn the camera to look to the left, you'll see a series of explosions in the air above the trees that looks a lot like flak. Now, these explosions are just gratuitous, and I added them just because the module can.



8.76.2.2 ME
So let's see what we got:

The boom stack
In the mission "Object Destruct Detector" we built an entire mission just to make you destroy something and show you that we could detect it. Well, now we stacked a bunch of modules on top of a trigger zone to do everything for you:



- The first four attributes ROLE; VALUE, OBJECT ID and NAME are auto-created by ME when you right-click on a scenery object, and are automatically consumed by the ODD module to watch this building. We simply added an output "destroyed!" that changes flag named "boom" when the building is destroyed.
- Next we see a raiseFlag! attribute that shows that we also stack a raiseFlag module on this zone. It's the timer that will tell groundExplosion module when to blow. When the timer runs down afterTime=5 seconds, the module raises the local flag "*go"
- And finally we come to the groundExplosion module also stacked on this zone that will create an explosion with the strength of 1000 at the center of this zone when the local flag "*go" changes its value

In short, we have set up an explosion that is triggered by raiseFlag after 5 seconds and that will blow up the building, hereby causing the OOD to send a signal on flag "boom".

Yes, we can achieve the same result by simply setting the "boom" flag with raiseFlag itself, but that would be so much more boring

The Fire Truck Ballet
These are just eye candy, and just serve as a nice show case how the destruction of a building can be used to trigger a couple of cloners across the map to cycle. When we look at one of them, there are no surprises:

Name	Value	
cloner		
clone?	boom	

All cloners cycle when there is a signal on the 'boom' flag, which is sent when the office building is destroyed (when ODD detects that the building it is watching has been destroyed). Upon cycling, the cloner produces a clone of the fire truck that has its route carefully arranged and choreographed with the other trucks

Oil Tank Ripple Explosions
Now let's look at the series of explosions that rip through the oil tanks. While the groundExplosion module we placed on the office building happened at that zone's dead center, here we have series of explosion that happen in random locations all across that (quad based) trigger zone.

Name	Value	
explosion	5	
boom?	boom	
num	20	
duration	5	

- The small-scale (explosion=5) detonations are started by a signal on the 'boom' flag which connects to the explosion? Input. So the destruction of the office building starts the series of explosion on the other side
- Since the 'num' attribute is greater then 1 (one), the ground explosion module automatically switches into randomized location mode, spreading the explosions all across the trigger zone that defines the groundExplosion's borders
- The 'duration = 5' spreads the series of num=20 explosions over a time of 5 seconds, creating the ripple effect

Flak!
This is just a convenient demo of further groundExplosion features that also somewhat belie the module's name. The Flak effect is created by

Name	Value	
explosion	2-5	
num	50	
duration	10	
AGL	50-70	
boom?	boom	

- a small-scale explosion that is randomized in power between 2 and 5
- num=50 explosions in total
- spread over a duration=10 seconds
- all occurring at a height between 50 and 70 meters (AGL=50-70)
- and triggered when the flag connected to the boom? input sends a signal. That flag is named "boom" as a cautionary tale not to use flag names that can be easily confused with input names. Or to show the opposite, depending on your preferences.

8.76.3 Discussion

The groundExplosion module es very versatile, and can be used to destroy objects (including map/scenery objects) on command, as well as spectacular, randomized explosive ripples and flak-like explosions in a volume of air.

It was one of the most requested modules in 2023

I still think I should have named the entire module ‘boom’.

8.77 Landing Lessons (TDZ)

8.77.1 Demonstration Goals

TDZ is a module specifically designed to help mission creators to create training missions, particularly training missions for landing aircraft. As the name implies (if you look at the long-form expression for TDZ which is “Touch-down zone”, the module can be especially helpful if you are designing missions that train people to touch down within certain areas of the runway, perhaps as training for a carrier deck.



8.77.2 What To Explore

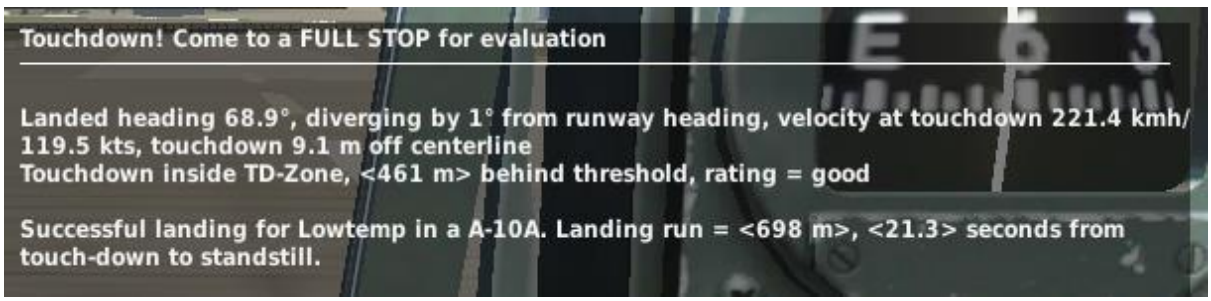
8.77.2.1 In Mission

Take control of Hog “Kobu A-10” (or switch it to any other aircraft that you prefer. Hit pause, and then go to F-10 map view and zoom in on Kobuleti. Note the two green zones on the runway:



Your goal is to touch down inside those green zones, as close to the center of those zones as possible. Switch back to cockpit view and unpause.

You are on long final for Kobuleti 07. Ignore procedures, and land the plane. Try to touch down close to the APC parked to the right side of the runway. After touch-down, come to a complete stop without ever leaving the runway. You’ll see something similar to



Note the “Touchdown inside TD-Zone” and rating bit. This will vary depending on how closely you touched down to the APC. If close enough, you’ll get an ‘excellent’, if too far, a ‘poor’, or (at worst) no rating at all.





Now switch to “Batu A-10R” or “Batu Hornet R” (again, you may want to change the mission to an airframe that suits you). Pause the mission, and switch to F-10 Map view and zoom in to Batumi. Note that there are two colored zones on Batumi’s runway: a green (towards the western end of the runway) and a blue one (eastern). Note also that these zones are *small*, each one marked by an APC for visual cues. Note that the green and blue zones have different offsets from their relevant runway thresholds (measured from the threshold in landing direction).

Now switch back to cockpit view, and unpause. You are on final for Batumi 13. Try to touch-down in the green zone close to the western APC, and come to a full stop. You’ll see similar messages to before, and if you are *really* good, you’ll also have managed to touch down inside the green TDZ.

Switch to “Batu A-10L” or “Batu Hornet L” and try to touch down inside the eastern (blue) zone (marked by the other APC).

8.77.2.2 ME

This demo’s main attraction are the two separate TDZ at Batumi, so we’ll focus on them. Let’s turn to “Carrier Landing Right” first, to face the ugly parts head-on. To begin work on a runway that uses two TDZ with different settings, make sure that you place them one by one, and turn off their runway frames when you are done, else their helper lines will get into each other’s way.

Now, this TDZ represents the green area, close to the threshold of Batumi runway 13 (i.e. the 130° landing heading for runway 13/31). In DCS, all runways are defined by a location and heading pair, with the reverse course being implicit – meaning that in DCS there is no DB entry for the runway for the reverse course. In the Caucasus map, Batumi is an airfield that has a single runway, heading 310 (to find out which runway has which heading as per Map DB, turn TDZ’s global verbosity on in the config zone). The TDZ module takes this, information and adds another runway in the opposite direction. To be able to differentiate the two, it calls the ‘original’ heading ‘left’, and the opposing direction ‘right’.

We use this mechanic in order to be able to create two separate TDZ for runway 13/31, one ‘left’ (following DB direction 310°) and a separate one ‘right’ (310°-180° = 130° → RW13). The TDZ is set up accordingly:

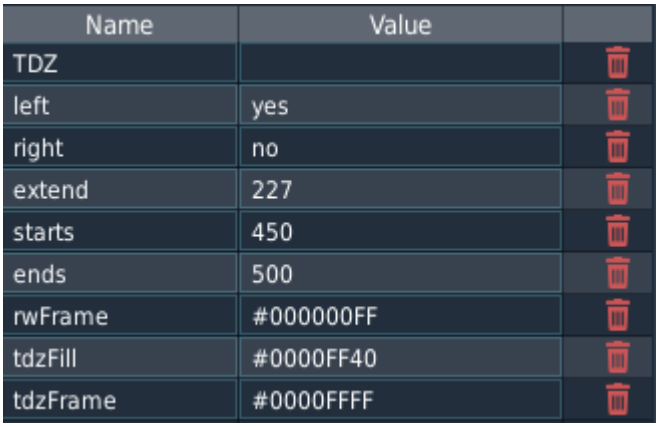
- the `left` attribute is set to false (original heading 310 is disabled), and
- the `right` attribute is enabled. This sets up this particular TDZ as one that is associated (by the trigger zone’s proximity to

Name	Value	
TDZ		
left	no	
right	yes	
extend	150	
starts	450	
ends	500	
rwFrame	#00000000	
helos	yes	

Batumi airfield) with RW31/13, but only in “right” direction, meaning that it only detects a touch-down in the zone that is set up for the 130° landing heading, close to that landing threshold.

- `extend` is set to 150 (meters) to make the entire runway length 150m longer on both ends (a total of 300m). How did I come up with this value? I ran the mission, and looked at the black outline of the runway that TDZ automatically draws. I then used the ruler too to measure the distance from the dotted outline to the F10 map’s true threshold of runway 13 – which came down to 150m. Make sure that you **set your units to meters in the ft=>m menu on the top** before you take down measures!
- The values for `starts` and `ends` are similarly measured with the ruler, making the TDZ exactly 50 meters long – which is still larger than the area that you must touch down inside on a carrier! The area is designed to enclose the runway’s Touch-Down markings (two fat vertical bars).
- Once I was done with measuring this runway, I set rwFrame to #00000000 (Red = 0, Green = 0, Blue = 0, Alpha = 0), a completely transparent color so TDZ does not draw the dotted outline for this TDZ (Carrier Landing Right)

We then turn our attention to the TDZ “Carrier Landing Left”, which should now be much easier to understand:



- `left` is not true, as this TDZ is facing Batumi’s ‘original’ DB heading (as we found out by setting verbosity to true and running the mission)
- `right` is set to false, because we want this TDZ only to service the original heading 310 (the other TDZ is servicing the ‘left’ direction, remember?).
- `extend` is set to 227 meters – which moves the runway threshold to RW31’s true threshold as we found out by measuring with the ruler tool. This means that RW13/31 is built asymmetric – which is true for many runways, and using two TDZ for the same runway can take care of that for you.
- `starts` and `ends` are again measured by the ruler to place a 50m long area over the fat two stripes that mark the touch-down area for this heading (310°).
- `rwFrame` is set to deep, fully opaque black
- `tdzFill` and `tdzFrame` are set to blue – 30% transparent and fully opaque to mark out this TDZ in a different color, just to show that you can.

8.77.3 Discussion

TDZ’s ability to mark out the TDZ on the F10 map is a nice QoL item, but for ‘production’ missions I recommend that you turn them off, and replace them with your own, better looking markings. Since these markings are only visible in F10 view, you should also think about marking their center with visual cues – like the APC I placed on next to the runway.

8.78 Player Score to Win (playerScore)

8.78.1 Demonstration Goals

This mission demonstrates how we can use PlayerScore (or more accurately, the module's direct outputs redScore# and blueScore#) to drive win conditions and messengers to announce scores instead of PlayerScore's

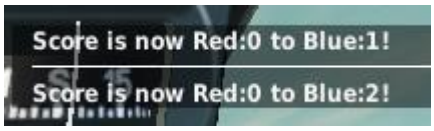
8.78.2 What To Explore

8.78.2.1 In Mission

The mission is set up so that the third kill wins the mission. Kills are announced individually and when blue scores the third kill, a message is put on the screen.

Enter the cockpit of your favorite plane ("Hogger" or "Frogger"), arm your weapons, and loosen them on the hapless red forces arrayed in front of you.

After two hits, you'll see something similar to



Note the increasing score count and the fact that this is very different from the way that PlayerScore usually would announce kills..

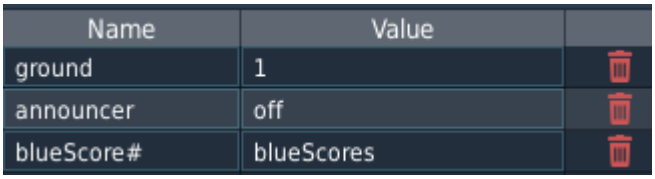
The third hit results in something like this:



Note the "Blue Wins" message.

8.78.2.2 ME

To use PlayerScore as a simple kill counter, we use the playerScoreConfig zone to set up PlayerScore as a silent, no-frills kill counter. It *can* do a lot more, but this nicely demonstrates how with only a small bit of creative configuring, even highly complex drop-in modules can be used to accomplish things that they don't immediately appear to be designed for:



- Setting `ground` to 1 sets PlayerScore up so that every ground unit kill yields a score of 1, making it a pure kill counter
- We set `announcer` to false ("off", "no", "0" and "false" are all internally translated to false when DML expects a true/false value like for `announcer`
- We also directly put the current value of BLUE's score onto the flag named "blueScores" by connecting it to the direct output `blueScore#`.

And that's already all (for this little demo). PlayerScore now adds the score for all ground kills, and this score coincides with the number of (enemy) ground unit kills (PlayerScore still filters blue-on-blue kills, as part of its default behavior).

The score announcements are made by a messenger that is triggered each time that the flag named "blueScores" changes its value:



- `messenger?` triggers on a value change of the flag named "blueScores", which is connected at the other end to PlayerScore's direct output "blueScore#"
- Whenever the messenger is triggered, it constructs the dynamic output `message` from the string "`Score is now Red:<v:redScores> to Blue:<v:blueScores>!`", replacing the wildcards with the current values, and putting the result on the screen

The advantage of this setup is that we can use messengers vastly superior wildcard abilities to create much better announcements. In theory, of course, because in this demo the text is a bit anemic.

So how do we declare the winner? Here we are using a plain vanilla Mission Editor trigger rule to show that even though inferior to "the DML Way", they still can be useful in a pinch, and are easy to integrate with DML



The rule triggers once on the condition that the flag named "blueScores" is greater than 2, and then sends a message to all that "Blue Wins!".

8.78.3 Discussion

Challenge: make the announce shut up when winning is announced, and increase the count to 10

Make the mission also usable for red, so they compete who is the first to 3

Only set up to count ground unit kills. Make it count all kills as 1

Make the mission end when one side reaches 10 counts and declare that side as the winner.

8.79 Clone Factory (CloneZone with Factory)

8.79.1 Demonstration Goals

While factoryZones have the ability to produce (spawn) units by themselves, you can also use the signals that they produce to drive cloners (or other modules) to act when a production cycle occurs.

This demo shows how you can use cloneZones to drive a factory's production. Cloners have some advantages (better, visual control over which units to produce and which path to take, formation etc) and some disadvantages (no ability to make the units attack the nearest enemy/neutral owned zone)



8.79.2 What To Explore

8.79.2.1 In Mission

Start the mission. There are no planes to enter, you simply watch.

When the mission starts, switch to F10 map view. South of Senaki-Kolkhi there are two factory zones, one owned by blue (west) and one by red (east).

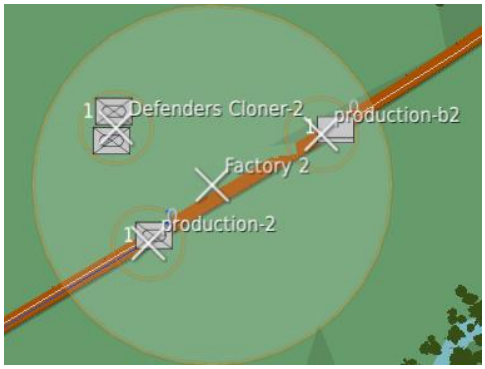
Let the mission run (perhaps accelerate time if you don't like to wait), and bother factories first produce defenders, and then produce a stream of vehicles that are heading for the other factory. Now, production for west and east is asymmetric: the blue factory produces Leo MTB, while the red factory produces BTR-80. Predictably, the BTR-80 don't stand a chance and are destroyed while the Tanks proceed to the red factory.

When the tanks are in range of the red factory, thy not only destroy all newly produced BTR-80, they also destroy the defending red AAV. As per factory rules, the factory then switches priority to producing defending units and only once the defenders have been reinforced, will the factory switch back to producing BTR-80.

After a brief but decisive battle, blue takes the eastern factory. Once captured, that factory first produces defenders, and then starts producing LUV Tigrs that head towards the last red owned zone in the north.

Once they reach and conquer the zone, the Tigrs head further north, and (more importantly), all production in all Factories ceases.

8.79.2.2 ME
All the differences between a factory stand-alone version and this mission are under the hood. Let's have a look at "Factory 2" – the eastern one that produces the BTR-80 and is quickly captured by blue:



In Mission Editor we immediately notice that the factory is home to a couple of other trigger zones, trigger zones that turn out to be the home of cloner zones:

- Defenders Cloner-2
- production-b2
- production-2

Usually, factories do not need cloners to produce, this demo specifically uses cloners to produce units, while the factory itself is configured to produce nothing but signals to drive other modules – cloners in our case

The factory is set up as follows:

Name	Value
owner	red
factory	
redD!	defend2
blueD!	defend2
redP!	produce2
blueP!	produceb2
defendMe?	helpMe2

- *owner* is red (remember that all factories must also be owned zones, else they won't work)
- There are no unit type strings that define what units to produce (*factory* itself is empty, and there are no 'production' nor defender attributes. This means that whenever a production cycle occurs, the factory itself does not spawn any units
- There are four outputs (*redD!*, *blueD*, *redP!* and *blueP!*) that each fire when a production cycle for that faction starts. They all connect to flags that (as we'll see soon) connect to the various cloners inside the factory zone that do the spawning.
- The *defendMe?* Input connects to the flag helpMe2. It connects to the clone zone "Defenders Cloner-2" *empty!* output.

The Factory 2 zone uses different cloners to produce units. Let's look at "defenders Cloner 2" first:

Name	Value
cloner	
masterOwner	Factory 2
clone?	defend2
preWipe	yes
empty!	helpMe2

- defenders Cloner-2 is shared between red and blue. It clones the two AAV-7 whenever it gets the signal on it clone? input, which is connected to the factory's *blueD!* and *redD!* outputs.
- An important bit is the *masterOwner* attribute that makes the cloner inherit its owning faction (and therefore ownership of the units it clones) from the "Factory 2" trigger zone. That way it always produces units that belong to the same faction as the factory itself.

- Also, the cloner's *empty!* output is wired into the factory's *defendMe?* input. This causes the factory to undergo a defender production cycle whenever the defending units are all destroyed, which will in time then generate an output on the factory's *xxxD!* flags

Then there are two different cloners for the "standard" factory production, one for red and one for blue.

Name	Value	
cloner		
masterOwner	Factory 2	
clone?	produce2	

- **production-2** uses *masterOwner* to inherit the owning faction from the factory zone itself
- The zone clones whenever it receives a signal on *clone?*, which is connected to the factories redP! output. This means that it will only be cycled when the factory is owned by RED. Looking at the units that are cloned, we see that they all are BTR-80 and have a route towards the western factory.

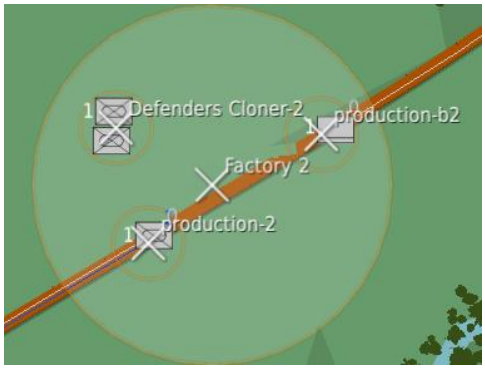
Name	Value	
cloner		
masterOwner	Factory 2	
clone?	produceb2	

- cloner **production-b2**, is very similar: it derives ownership from the "Factory 2" zone
- It cycles clone production every time that it receives a signal on the *clone?* input via the flag named produceb2. Looking at the factory zone, we see that this is wired into the *blueP!* output. This means that the cloner can only cycle when the factory is owned by BLUE

8.79.3 Discussion

They are all neutral!
One of the first things that you notice is that the units inside the spawners are all neutral. This is for two reasons:

- Since the cloner inherits its faction from the factory, this is a great way to indicate to yourself that the faction for these units isn't fixed when you design the mission.
- More importantly, though, a slight irregularity within DCS (especially with airfields) can cause issues when you set up a cloner inside an owned zone: the clone template may briefly confuse the owned zone when it determines which faction it belongs to during mission start-up (this only can happen with airfields, or when the cloneZone module starts after ownedZones): for a brief moment it can look as if enemy units are inside an owned zone, which may even trigger a conquered event.



So in order to not upset zone ownership, we define clone templates wither outside of owned zones and import the template later, or we use the NEUTRAL faction when we set up the cloner's template. Since the cloner inherits the faction from the owned zone, they will be spawned as members of the correct faction.

It ends without a bang!

When blue conquers the red factory, it starts producing LUV Tigrs that head northward and, a shot while later, capture a small ownedZone. This is intentional design: if we didn't do this, all production would have ceased the moment that blue had conquered red's factory. Why? Because factories only produce units as long as there are neutral/enemy owned zones.



So by adding this red owned zone we ensure that Factory 2 continues to produce units after it was captured (so we can test the ability to produce different units for red and blue), and only when the blue Tigrs then capture that owned zone will both factories stop producing units: there is nothing left to conquer.

We achieve stopping all production without ever banging on the factory's '*pause?*' input flags (which we don't use in this demo)

8.80 Airfield Mine (airfield, cloneZone, messenger)

8.80.1 Demonstration Goals

This mission demonstrates how you can use the airfield module to report changes in ownership, drive cloners to spawn units for different factions, and change ownership of an airfield at the drop of a hat (figuratively speaking, of course)



8.80.2 What To Explore

8.80.2.1 In Mission

Start the mission, and enter your trusty frogfoot. It's tucked away safely in Kutaisi, no nothing untowardly can happen to you.

Press F10 Map view, and zoom into Senaki-Kolkhi. Note the collection of NEUTRAL units: a couple of aircraft, a vehicle, and a single soldier (off the west end of runway 9). There is a blue IFV Marder approaching Senaki's 2km conquer radius.

When it gets clone enough, a message comes up



and, more importantly – all units on the airfield (those that were there originally as NEUTRAL) turn blue!

Now go to communications→Other…→Set Senaki Ownership and choose "Give Senaki To Red".

A message comes up



And all those units turn red – while the Marder is still approaching. A short while later, the Marder will have destroyed all red ground troops (the AK and fuel truck) – yet the airfield remains in RED's possession, which is clearly against standard conquering rules (the Marder should capture the airfield).

Under communications→Other…→Set Senaki Ownership choose "Whoever Can Capture", and a few seconds later, Senaki again belongs to blue (the Marder captured Senaki) and a corresponding message appears – along with a fresh set of blue units.

8.80.2.2 ME

The real star of the show are the aircraft and vehicles that apparently change their faction with the airfield:



Of course, that's merely a clever illusion, and we'll investigate how we did this so you can use this and similar approaches in your own mission.

First, we notice that all these faction-hopping units are inside the clone zone "Owned Cloner", and this clone zone uses the following properties to create the effect of faction-hopping:

Name	Value
cloner	
clone?	senakiOwner
preWipe	yes
masterOwner	Senaki Owner Control
onStart	yes

- *masterOwner* tells the cloner to get the ownership for all clones from the zone "Senaki Owner Control", which is a zone with managed ownership (in our case an airfield Zone)
- *clone?* is triggered by a change in the flag named "senakiOwner", which we'll find is fed by the airfield's *ownedBy#* direct output – each time that the airfield changes hands, the cloner is triggered.
- *preWipe = true* makes sure that all previously cloned units are wiped; when the cloner then spawns the new units, they belong to the new owner, making it appear as if the units changed ownership.
- *onStart* ensures that there are units (NEUTRAL) when the mission starts up

And now let's focus on the other attraction of the show: being able to force the affiliation of an airfield. This is one of the inherent abilities of an airfield zone, and we drive this ability by with the outputs from a convenient radioMenu

Name	Value
airfield	
makeRed?	makeRed
makeBlue?	makeBlue
ownedBy#	senakiOwner
autoCap?	anyone

- *makeRed?*, *makeBlue?* and *autoCap?* are all fed by flags that come directly from a radio menu and set the airfield's coalition directly (*makeRed?*, *makeBlue*?) or allow it to revert to classic capture rules (with *autoCap?* – remember that once you force an airfield's coalition, it remains fixed until you explicitly allow capture rules)
- the direct output ownedBy# is used to drive a messenger module that outputs a message every time that the airfield changes hands.

8.80.3 Discussion

Name	Value	
messenger?	senakiOwner	
message	Senaki now is owned by <cc	

In case you didn't notice – the messenger also has a cool trick up it's sleeve: it is triggered every time that the airfield zone reports a different owner (which triggers the message itself), and the message uses the <coa> wildcard to fetch the current owner from the airfield zone and displays it as current text:

```
Senaki now is owned by <coa: senakiOwner>!
```

Which in-game translates to



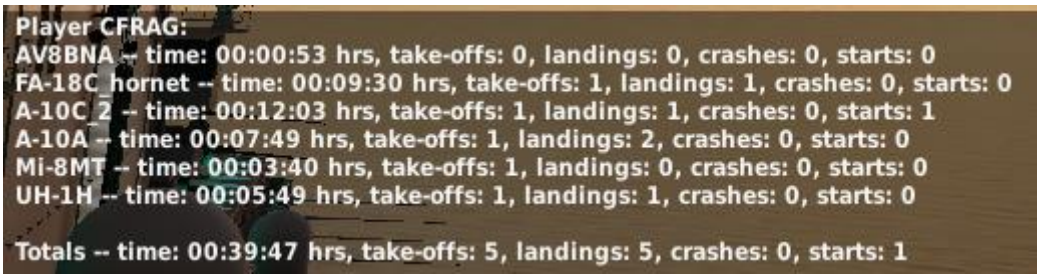
and



8.81 On The Record (Scribe, Scribe + csarManager)

8.81.1 Demonstration Goals

This demo shows the principal workings of the 'bean counter' module Scribe, and how it integrates with csarManager.



8.81.2 What To Explore

Note I
There can be great differences in the experience that you get when playing this mission depending on how you set up your DCS installation. If you have 'de-sanitized' DCS (meaning that you are allowing DCS missions to write to storage), you will likely get different results than if you left DCS as it was installed. Refer to the 'persistence' section of this manual on details of 'sanitizing' and 'de-sanitizing'.

Note II:
This demo mission requires some effort on your part and may exceed your abilities and/or modules that you own. To follow this demo, I expect that:

- You have access to another aircraft other than the free Su-25
- You can cold-start, take off and survive a landing of one of those two planes
- You have access to a helicopter (I'm using the Huey, change it to any helicopter that you have access to)
- You can fly that helicopter well enough to precisely land where you want to at least two times in a row (you need to pick up an evacuee and return them to base)

Note III:
Parts of this demo assume that you have understood the objects and workings of the demo missions CSAR of Georgia (to understand how DML gives you easy access to CSAR missions) and Being Persistent (to understand persistence in DML)

Of course, if you do not (yet) qualify for all the mission's demands, just read through the description and dissect the mission itself.

8.81.2.1 In Mission
Start the mission for the very first time and enter Frogfoot "Frogger COLD". You'll be greeted with something similar to



Cold-Start the Frogfoot (a simple affair involving pressing “electrical power switch” followed by moving throttle to idle (!) and then hitting ‘Engines Start’ and getting a coffee while waiting up to 3 minutes).

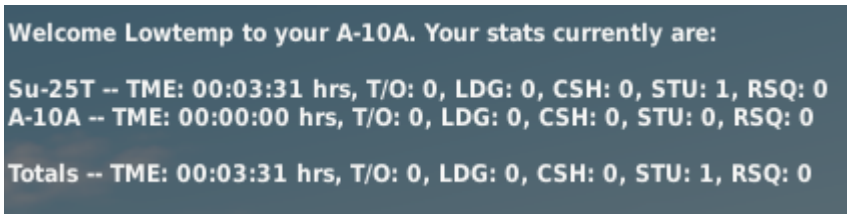
Once the engines have started up, go to communications→Other→Mission Logbook and choose Get Pilot’s Statistics.



Note that STU has changed from 0 to 1.

Once start-up is complete, taxi and take off, then land the Frogfoot. Note how the statistics for T/O and LDG also change: T/O increases by one for each successful take-off, and LDG increases with each successful landing.

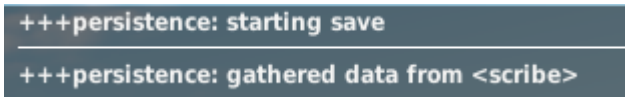
Now change into a different airframe. I’m choosing the HOT Hogger (A-10A). As soon as you enter the Hog, you are greeted by something similar to:



Note that the statistics displayed have expanded and include your time in the Frogfoot (and where I obviously did not bother to take off nor land, just made you do it for demonstration purposes). Also new is a Totals line that adds up all your various accomplishments and time spent in the various airframes.

Start the Hog, and once successful started, check the Logbook to ensure that you have been awarded a 1 for STU in the A-10A as well.

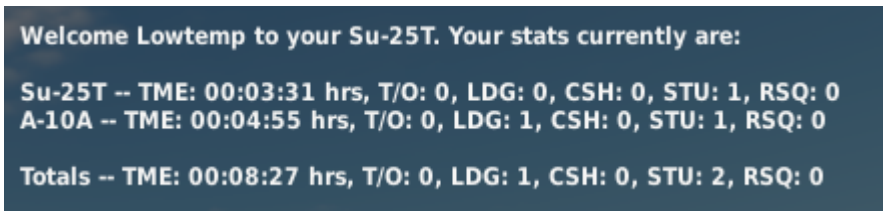
Now, go to Communications→Other→Save Mission and choose “Save Now”. You’ll see



plus some more details where persistence saved to.

Now exit and end the mission.

Start the mission again, and choose the Hog or Frogfoot (I chose the Frogger). If you have successfully de-sanitized DCS, your welcome message is different from the first time:



It includes all your accomplishments from missions past (that have saved).

Now, jump into a helicopter (I'm jumping into the Huey).

Go to F10 Map, and zoom so that you can see both Senaki-Kolkhi's airfield and the town proper to the north of the airfield. Note that there are three Infantry M4 inside the city property. These are the CSAR missions that are created at mission start, and their locations are randomized.

From Communications→Other→CSAR Missions choose "List Active CSAR Requests" to receive a list similar to this:



Note that the names that you receive will be very different from above, as well as their bearing, ADF frequencies and distances.

Now, select one (or more) of the evacuees, fly to them, land, and then return with them to Senaki-Kolkhi and land on the tarmac area, at which point the SAR mission(s) complete. Look at your Logbook



The RSQ ("rescues") entry now reads "1" for me after rescuing one of the evacuees.

8.81.2.2 ME

Even though complex inside, most of what this mission demonstrates happens entirely automatic and completely transparent to you, the mission designer. Let's start with the most obvious: the logbook and how your achievements are listed.

First of all, all record keeping is automated and happens behind the scenes as soon as you add the Scribe module. It simply can't help itself, and records everything players do.

Next the way that scribe reports: the report has an unusual format. Instead of the column heads "*Landings*" etc. we see some strange three-letter abbreviations like "LDG".



Name	Value	
lTime	TME:	
lLandings	LDG:	
lDepartures	T/O:	
lCrashes	CSH:	
lStartups	STU:	
rescues	yes	
lRescues	RSQ:	

This is set up with scribe's config zone: we use the *lXXX* attributes to change the column names. I chose three letter abbreviations in order to make some room for the last, and new column that usually (by default) isn't displayed: RSQ for rescues. The *rescues = true* attribute tells scribe that this data is to be reported (default is false), and with *lRescues = RSQ:* (note the colon ":" at the end!) we set the new column's name to be "RSQ:".

Behind the scenes, Scribe fetches the rescue data from csarManager automatically, this does not require anything from you to enable.

So, on the surface, this concludes our discussion of this mission that shows you how to use scribe in your mission, and get it to report your helicopter rescues.

Now, if you are interested in the really cool stuff, read the Discussion.

8.81.3 Discussion

Wait… ALL Helicopters?
The first thing to mention is something that you may only have discovered if you had tried to play this mission in a helicopter that is not supported by csarManager out-of-the-box. Here, suddenly, csarManager supports *all* helicopters, even some types that DCS doesn't know about (for example the user-created Blackhawk). This is due to the fact that I added a config zone for csarManager:

The *troopCarriers* attribute can be used to tell csarManager which units can perform rescue missions, and setting it to the special *"helos"* keyword allows all helicopters to perform them.

Name	Value
addPrefix	no
troopCarriers	helos

Don't be a downer…
There's another attribute here: "addPrefix = no" – it is used to shorten an evacuees name. By default, csarManager would prepend 'Downed' to a pilot's name, since the most common use case for csarManager is to rescue downed pilots. Here, we omit that and only show the names of the people to rescue, without the 'downed' prefix.



Jessie Powell? Ferdinant Barker? Nice names!
If you continue rescuing people you will notice that each mission sports their own evacuee name, and (unless you are *really* dedicated) they don't repeat, yet sound realistic. csarManager uses the 'random names' feature here that accesses the **names** module to generate (mostly western) names from a database of common names and surnames that can generate up to 360'000 unique names. We turn on this feature by setting **rnd* as value for the *csar* attribute.

Name	Value
csar	*rnd

Wow, those missions surely are conveniently placed on roads…

onRoad	yes

After rescuing a couple of evacuees you will have noticed that the people that you need to evacuate are all placed nicely accessible on a road. This is no coincidence and controlled with the "*onRoad* = *true*" attribute for the CSAR missions. This option is incredibly helpful if you want to create accessible (in a very literal way) randomly placed missions (remember that CSAR missions default to random locations inside their zone). In this mission, the randomized area is a good chunk of Senaki-Kolkhi's city proper, and if we merely chose a random location inside the area, there would be a good chance that the location comes to lie inside a building, making the mission impossible to complete.

Since roads are much better accessible, and (at least on the Caucasus map) wide, telling csarManager to place evacuees on roads is a good compromise. The downside is that this only works if the region where you place CSAR missions needs to have roads. Remember to turn off road traffic lest you evacuee gets run over, or your helicopter become road kill during the rescue attempt.

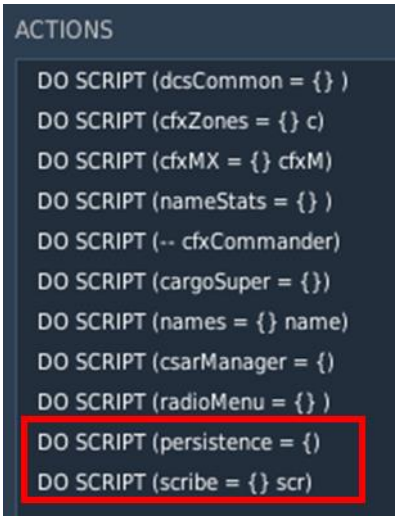
Huh. Only Scribe persists its data?
This is something so subtle that very few people notice outright: although the mission has persistence, only Scribe persists its data. This is intentional (of course) and a result of taking advantage of a persistence rule that is easy to overlook:

A section on persistence in this manual states (even in bold face):

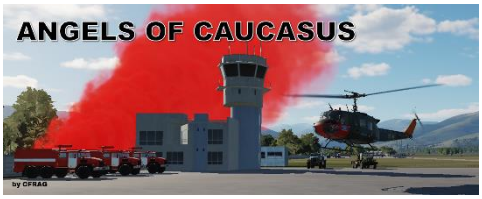
Persistence is only available to those modules that load after persistence itself loads.

The reason behind above rule is that modules must sign up to persistence, and if persistence isn't loaded when a module starts up, it can't sign up. If a module hasn't signed up for persistence, it won't get asked for data to save when persistence does its thing, and consequentially there is no data to load when persistence loads on the next start up.

We use this to our advantage and make Scribe the *only* module that loads after persistence. That way, only Scribe subscribes to the persistence services, and only the player accomplishments are persisted between mission runs.



Wait… haven't I've seen this mission before?
In case you were wondering: yes, this demo is 80% of Angels of Caucasus – except that I moved it to Batumi, used multiple CSAr zones, a randomizer and polished some more. But this demo is the essential prototype for many city-based rescue scenarios.



8.82 Civvy Heli

8.82.1 Demonstration Goals

This mission shows how you set up a mission to have AI helicopters randomly fly between points on the map.



8.82.2 What To Explore

8.82.2.1 In Mission

Start the mission, switch to F10 map view, and lean back – there is nothing to fly for yourself, all you are expected to do is watch.

Zoom out slightly so that you can see all four red points marked “Upper City”, “Lower Meadows”, “Lower City” and “Senaki Airfield”.

After a short while, you’ll notice neutral (white) helicopters spawning at two of the red circles (these spawns are randomized, no way of telling which two points it will be, nor what the type of the spawning helicopter will be).

Use F2 to briefly inspect the helicopters and note their type and livery. The helicopters will take off, fly to their destination, and land there.

After a while (you may want to accelerate time) you may see the following patterns emerge

- Whenever a helicopter spawns at Senaki Airfield, it is a Huey (UH-1)
- There never spawn two Helicopter at once at Senaki Airfield
- The Huey at Senaki Airfield always starts cold, all the helicopters at other locations start hot
- There’s never a flight that goes TO Lower Meadows, but flights come FROM that location
- The only helicopter types that spawn are UH-1H, Mi-8MT and SA342L
- Helicopters can simultaneously take off and land at Senaki

8.82.2.2 ME

Only Three Types

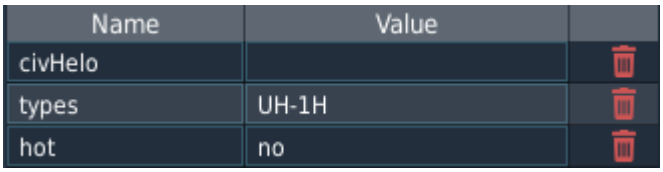
In this mission, civHelo only uses three types of helicopter: Hueys, Hips and Gazelles. This is controlled by the

Name	Value
types	UH-1H, Mi-8MT, SA342L

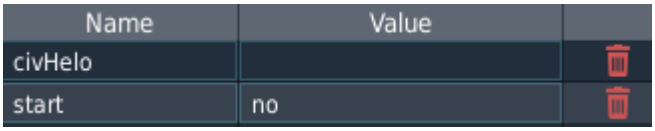
civHeloConfig zone that limits all types that are used to these three.

Only cold Hueys start at Senaki
There are *two* civHeloZones at Senaki Airfield, and only one of them (the northern one) can spawn helicopters, while helicopters can land at both zones. All this is controlled by the zones Senaki Heli-1 and Senaki Helo-2.

Senaki Helo-1 is the more interesting of these two: it tells civHelo under *types* that only UH-1 (Hueys) may start here, and that the helicopters that start here should cold-start.



Senaki Helo-2 is also interesting only because of the fact that it is very close to Senaki Helo-1, and does not allow helicopters to depart from here.



This means that there can only ever depart one Helicopter at the same time from Senaki (a Huey from Senaki Helo-1), while it is quite possible that two helicopters can be arriving at Senaki Airfield at the same time: one at Senaki Helo-1 and one at Senaki Helo-2. Since both zones are very close (less than minDist apart) together, civHelo will never generate a flight between Senaki Helo-1 and Senaki Helo-2.

Note that although we are using heli pads for all our non-airfield locations, this is not required if the ground is flat, so you do not have to place those helipads. It looks better, though.

8.82.3 Discussion
CivHelo is a very easy to use module that can generate attractive local helicopter traffic for your missions. There are some other, unexpected features that you can use:

Challenge
There's an offshore oil platform off to the west. Add its helicopter platform as a departure-only source of flights, and make only Hips (Mi-8) depart from there.

Then see what happens if you allow flights to land at that platform.

8.83 Red vs. Blue / Blue vs. Red (shared data)

8.83.1 Demonstration Goals

Some modules (e.g. “Scribe”) can share data across missions. Usually, when a mission saves data, that data is written to a private data file (see the *persistence* module). Some modules can opt to instead to write their data to a different file in DML’s ‘shared data’ directory. These modules then can access that shared data and ‘continue to use’ the data, which can result in the illusion that the mission ‘continues’ from previous missions. For example, the “Angels” series of (freely available) rescue sandbox missions (Caucasus, Levant) all share the player’s achievements (like number of people rescued).



This demo mission shows how the scribe module uses the ‘sharedData’ attribute to achieve this.

8.83.2 What To Explore

This demo consists of two missions: “rvb BLUE” and “rvb RED”

Note:
In this mission we use the two free DCS aircraft Su-25T and TF-51 simply to be able to tell them apart. It can work much better if you substitute or add your own (owned) modules for those, but it is not required.

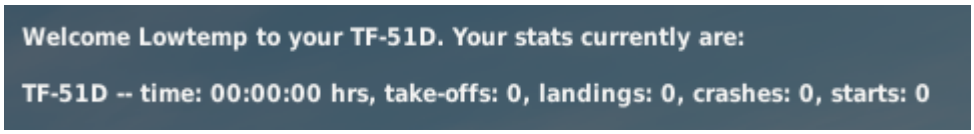
8.83.2.1 Preparation

When run, the missions access the directory “DML-Shared-Data” which by default resides in your standard DCS Missions/ directory. If that directory “DML-Shared-Data" exists, open it, and ensure that there is no file called “rvb.txt”. If it exists, delete it. This file holds the data that is shared between the two “rvb” missions, and to make sure that we get the same experience, delete the file to reset the data that is shared.

8.83.2.2 In Mission
Start with the BLUE side first: start “rvb BLUE”

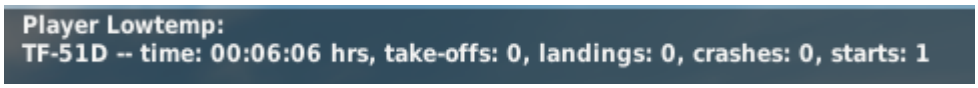
Enter one of the planes. Since everyone has access to the free TF-51, I’ll choose “Blue Mustang” even though I’d much more have preferred another one that is tantalizing me.

As soon as you enter the cockpit, you are greeted with



That is the *scribe* module, duly noting your arrival, and showing you your current statistics: zero total time, no take-offs, landings, crashes, nor start-ups.

Get oriented in the cockpit, and start the engine (if you are unfamiliar with this aircraft, use auto-start). Once the engine is running, go to Communication→Other→Mission Logbook and choose “Get Pilot’s Statistics”.



Your total time has increased, and you have one start credited to you. Look around some more, then go to Communications→Other→Save Mission and choose “Save Now”.

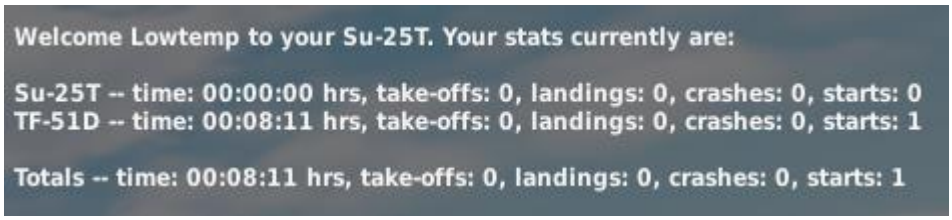


Persistence tells you that it saved the mission.

Optionally, on a re-run of this demo:
also enter any other plane that is available to you in this map.

Now exit the mission, and run “rvb RED”. Note that you are running this mission for the first time, that it has never before saved data.

Enter the Su-25T “Frogger RED”. You are greeted with something like



Note that *scribe* greets you, and that *it includes stats for time and other data that you have accumulated in the “rvb BLUE” mission*: your time in the TF-51 is accounted for, even though “rvb RED” does not even offer you to enter that aircraft! Your time and other stats for your current aircraft (the Frogfoot) are set to zero.

Start the Frogfoot (make sure that your throttle is on idle!), make sure that scribe has registered your start-up:

```
Player Lowtemp:
Su-25T -- time: 00:02:26 hrs, take-offs: 0, landings: 0, crashes: 0, starts: 1
TF-51D -- time: 00:08:11 hrs, take-offs: 0, landings: 0, crashes: 0, starts: 1

Totals -- time: 00:10:37 hrs, take-offs: 0, landings: 0, crashes: 0, starts: 2
```

Then save the mission the same way that you did before.

Exit the mission, start “rvb BLUE” again, and enter the TF-51 once more.

```
Welcome Lowtemp to your TF-51D. Your stats currently are:

Su-25T -- time: 00:03:33 hrs, take-offs: 0, landings: 0, crashes: 0, starts: 1
TF-51D -- time: 00:08:11 hrs, take-offs: 0, landings: 0, crashes: 0, starts: 1

Totals -- time: 00:11:44 hrs, take-offs: 0, landings: 0, crashes: 0, starts: 2
```

Your accomplishments from the mission “rvb RED” have carried over to “rvb BLUE”: your time and start-ups in the Frogfoot are shown even though there is no Frogfoot in this mission.

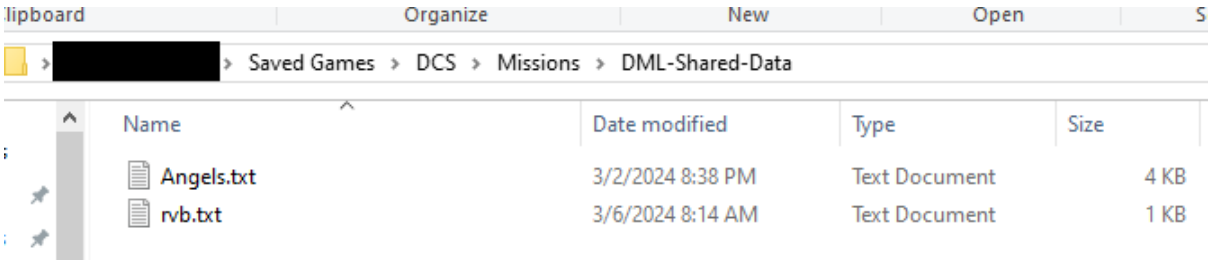
8.83.2.3 ME

The scribe modules in both missions “rvb RED” and “rvb BLUE” share their data: what it saved from one mission is read (and saved) by the other. We accomplish this with a single attribute in scribe’s config zone *scribeConfig*:



The *sharedData* attribute tells scribe that it should save and retrieve any data from a file called “rvb” that resides inside the “DML-Shared-Data” folder instead of the mission’s private data folder.

Note that both missions “rvb BLUE” and “rvb RED” must have configured scribe that way in order for both missions to share their data.



In the Image above, we see the contents of my “DML-Shared-Data” folder. There’s the file “rvb.txt” that is shared by the missions “rvb RED” and “rvb BLUE”, and another (not discussed here) file “Angels.txt” that is shared by other missions (my “Angels” mission series that currently comprises of “Angels of Caucasus”, “Angels of the Levant: Syria” and “Angels of the Levant: Sinai”).

8.84 Non-SSB and SP slot blocking

8.84.1 Demonstration Goals

The *Slotty* module provides “SSB” smart slot blocking for single-player games, and can serve as a fallback for multiplayer games if SSB isn’t installed on on the server. This mission demonstrates the effects of active slot blocking of player aircraft on airfields that do not belong to the player’s faction (the latter being provided by the *ssbClient* module).

This demo shows in single-player how a player-slot is blocked from entering while the airfield belongs to red, and the slot then becomes available after the airfield is captured by blue.

After that, the demo also shows how *slotty* provides a fallback to provide support for the SSB protocol when SSB itself is disabled or not installed.

8.84.2 What To Explore

8.84.2.1 In Mission

Start the mission in **Single-Player**. Before you can choose a slot, some of the typical DML startup diagnostics appear. Relevant for later is only one:



There are two slots available to choose from. Choose “Forbidden Frog”, the Su-25T located at Kutaisi airfield. Note that you can choose the slot. Enter the aircraft. Press “Fly” in the ensuing Briefing dialog.

As soon as the dialog disappears, so will the cockpit, and you are afforded a nice yet uneventful view of Kutaisi: you have been removed from the plane because it sits on Kutaisi airfield which is currently owned by RED and the plane that you entered is a BLUE plane. There’s also a not to that effect:



Now press ‘ESC’ to return to slot selection, and choose “Senaki Frog”. This aircraft you can enter, and after the briefing screen you are sitting comfortably at the controls of your Frogfoot.

Now Choose Communications→Other→Capture Kutaisi

Wait a few seconds until you see the message



appear among some others.

Now, press ‘ESC’ again to return to slot selection and choose “Forbidden Frog” again.

This time you can enter the Frogfoot at Kutaisi.

Now start the mission as **Multiplayer**

Go through the same steps and note that the results are the same: you can't enter the "Forbidden Frog" until you have captured Kutaisi

If you have installed SSB on your DCS also note that server-side SSB is disabled

8.84.2.2 ME

Since Slotty ans SSBClient work out-of-the-box there is very little going on in ME outside of adding the dependend modules, and setting the owners of the airfield. Kutaisi is to be owned initially by RED, Senaki by BLUE.

Capturing Kutaisi is done by activating a 'late activation' group of BLUE vehicles that are inside Kutaisi's capture circle.

8.84.3 Discussion



There is one tidbit worth mentioning and that is when we tested multiplayer: no matter if you have SSB installed on your DCS, it does not kick in for this mission. This is intentional, since we wanted to demonstrate Slotty's ability to work its magic in multiplayer as well. This is accomplished by SSBClient's config zone, which has set the ssbAutoenable attribute to false. This prevents sscClient from enabling SSB on the server, so it does not know that the mission is using the SSB protocol, and will not interfere.

8.85 What's on the cascading menu

8.85.1 Demonstration Goals

This mission demonstrates how you use trigger zones with 'radioMenu' und 'radioMainMenu' attributes to create "cascading" menus, i.e. menus in menus.

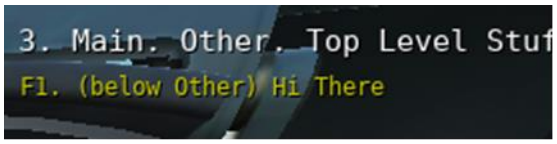
8.85.2 What To Explore

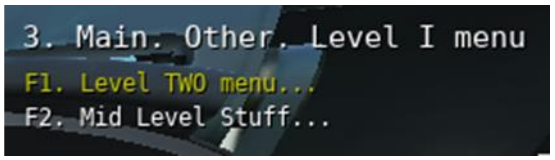
8.85.2.1 In Mission

There is no flying in this demo. Simply enter the Frogfoot and go to "Communications→F10 Other". Note the menu options:



First, choose "Top Level Stuff", then choose "Previous Menu" and choose "Level 1 Menu"





First choose "Mid Level Stuff", then "Previous Menu" and choose "Leven TWO menu"





Finally, choose "Bottom Level (2) Stuff"

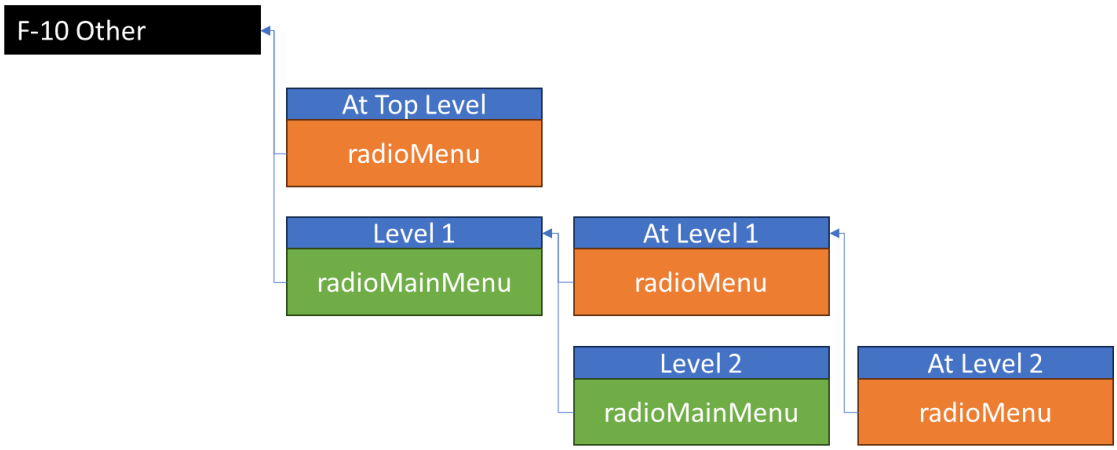


As you can see from the leading Numbers in the upper left corner we are progressing further down the menu tree.

The entire tree was constructed with DML-enhanced trigger zones that use either a *radioMainMenu* or *radioMenu* attribute.

8.85.2.2 ME

So let's look at this in Mission Editor. There are a total of 5 trigger zones that are used to assemble this (complex) menu tree:



There are three radioMenu (orange) trigger zones that provide their menu and items at the various levels, and two (green) radioMainMenu trigger zones that provide sub-menus where radio menus can plug into. Each radioMenu plugs into a radioMenu with an 'attachTo:' attribute, or (when there is not 'attachTo:' attribute) directly appears inside the F-10 Other menu.

8.85.3 Discussion

Remember tat the *attachTo:* attribute always refers to the *Trigger Zone Name*.



Don't become cyclic

It's difficult to do so, but people have tried: You can create cyclic 'cyclic' menu structures with "radioMainMenu" by referring from A to B and then in B referring back to A (or in more complex structure). Most often, when this happens, it appears when you are restructuring your mission. DML detects such an issue, and breaks a cycle, issuing a warning similar to

```
+++radioMenu: ERROR: Cyclic references in menu structure, can't fully process main menus.
Unresolved main menus are:
 Cyclic B, Cyclic A.
```

DML lists all trigger zones by names that are part of at least one cycle, and you should check these zones and break all cycles. Note that all menus that are part of that cycle will not work. Note also that cycles in your menu structure will not cause DML to freeze.

Challenge:
In the demo mission there are two trigger zones called “Cyclic A” and Cyclic B*. Their radioMainMenu attributes have been rendered inert by an inserted X.

To see how DML handles cyclic structures, remove the “X” from both, and run the mission. You’ll get the error shown above.

Now break the cycle, and insert Cyclic A and Cyclic B before “Level 2”.

8.86 Reaper, man

8.86.1 Demonstration Goals

This mission shows you how you can set up reaper drones to use with your missions – and use the built-in interface as well as a flag/radioMenu-driven 'homebrew' interface.



8.86.2 What To Explore

8.86.2.1 In Mission

Note:

To test target lasing provided by drones, you need a module that supports laser-guided munitions. This demo mission provides slots for the F-16 and F-5E. You may want to exchange this for your preferred LGB bomb truck.

Enter your trusty "The Watcher" (Su-25T) cockpit at Senaki Kolkhi.

Almost immediately, a message pops up:



Switch to F-10 Map, and note that there are 4 (four) drones in the air, looking for targets. The one in the North, East of Sukhumi is even tracking a target, the others are idling.



(Note: don't wait too long after entering the Frogfoot's cockpit, SAMs, especially SA-11 do not like to be spied on).

After a short while, you'll receive another message:



Pan your map, and verify that you can see the target that the Zugdidi drone is tracking. Note that red smoke erupts from the unit's location as soon as the drone has spotted the AAV7 and starts lasing. Note also that this drone is doing its rounds at 4000m altitude.

Now, if you have the module and skills, jump into your favorite LGB bomb truck, and drop a bomb on the hapless AAV7 that the drone is lasing. For easier orientation, look for the red smoke.

Now go to Communication→Other→Drone Command and choose "Drone Status". A report appears:



Note that there are 4 drones in the air, two of them track (and lase) targets, and one drone is ready to launch. Let's launch it.

Go to Communications→Other→Drone Command→Launch Drones and choose "unlaunched: MQ-9 Reaper".

Switch back to F10 Map view and note that a new drone is circling the skies above Zugdidi, slightly Northeast to the one that targeted the AAV7, at 9500m altitude.

At the 3-minute mark, the SA-11 have enough of being spied upon by a drone, and they fire at the drone:



however, this is immediately followed up by another



and looking at the map you'll see that the drone over the SA-11 site was immediately replaced. If you let the mission run, this pattern repeats: the SAM site downs the drone after a while, and a new drone arrives within a second.

Now let's turn our eyes to the Communication→Other→Secret Drone menu and choose "Status"



You receive a status of a drone that was not listed with the "Drone Status" command from the "Drone Command" menu. Let's launch it with Communications→Other→Secret Drone→Launch:



If you switch to the F-10 Map, you'll notice a new reaper drove circling above Batumi.

8.86.2.2 ME

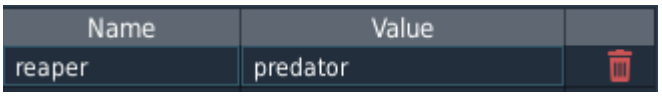
This demo mission shows you how you can quickly add drones to your missions and runs through the most common use cases.

Placing a drone on the map

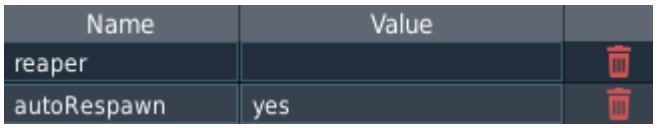
Look at the trigger zone "Kutaisi". It's the most basic reaper setup: a single *reaper* attribute with no value.



This sets up a drone using all defaults: an MQ-9 Reaper, launched at mission start, orbiting over the trigger zone at 9'400m (28'000 ft) using a laser code of 1688. Note that this trigger sone is quad-based.



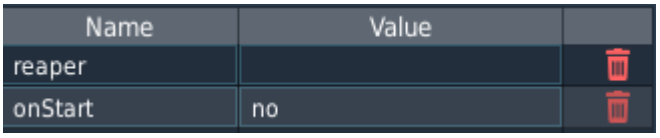
Slightly to the west we find the trigger zone "Honi", which is similarly simple, but holds the value "Predator" for the *reaper* attribute. This zoner spawns an RQ-1A Predator upon mission launch, which orbits the trigger zone at 7'500m (22'500ft).



As you probably have guessed by now, the self-respawning reaper drone monitoring the SA-11 site is similarly simply, and merely has an active *autoRespawn* attribute.



The trigger zone "Zugdidi" is more interesting: It spawns an MQ-9 Reaper (default) at 4000m, as defined with the *alt* attribute. Since visual range of drones depends on their altitude, this drone only has a visual range of roughly 4 km, which explains why it takes so long to detect the AAV-7. More importantly, though, this reaper zone is set up to give pilots more visual aids when the drone starts targeting units: it places a red (*smokeColor = red*) colored smoke (*doSmoke = yes*) at the point where it first spotted the unit (note that the unit may move after being spotted. While the laser will remain on the unit, the smoke marker remains at the initial location).



Finally, there's the "unlaunched" trigger zone which simply holds a drone ready for launch, but does not launch the drone on mission start (*onStart = no*).

All of these drones can be controlled with the "Drone Command" UI that the reaper module provides (for each side)

Note that all these drones can be launched with the Drone Command UI if they are not loitering/tracking targets. Also (not shown with this demo), if the bank module is present and this ability is enabled, the Drone Command UI automatically manages funds for drone launches, and only allows drone launches if there are sufficient funds available.

8.86.3 Discussion

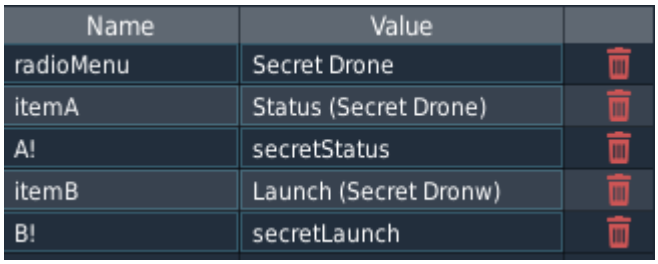


Sometimes, you may have more than 10 drones for a side, or you have drones that you do not want to be visible from the Drone Command UI. The reaper zone "Secret Batumi" is one such drone.

Its *launchUI = no* and *statusUI = no* attributes prevent it from appearing in the Drone Command UI. To still be able to launch or get a status for drones controlled by this reaper zone, we have the inputs *status?* and *launch?* that allow us to

inquire the status of the drone, and launch it on-demand (if it's not already launched). Since it is set to *onStart* = *no*, the drone does not launch on mission start.

To do this, this demo sports a radioMenu that supplies the inputs to get a status update and launch the drone.



Name	Value	
radioMenu	Secret Drone	
itemA	Status (Secret Drone)	
A!	secretStatus	
itemB	Launch (Secret Dronw)	
B!	secretLaunch	

8.87 Escort a convoy to town (Convoy & twn)

8.87.1 Demonstration Goals

This demo illustrates the use of the “Convoy” and “twn” modules to build convoys (Convoy) that are randomized and move purposefully from A to B – and have improved reporting (twn).



8.87.2 What To Explore

8.87.2.1 In Mission

Note:
Due to the heavy randomization built into the Convoy module, your experience of the play-through of the demo may diverge significantly from my description. Perhaps run through the demo a couple of times to understand what happens.

Start the mission in in the BLUE Frogfoot “Rubber Duck” and switch to F10 Map view. Go to Communications→Other→Convoy Control→Spawn Convoys and choose “Start new BLUE Convoy A”.

Southwest of Senaki Kolkhi a new blue convoy consisting of 3 BTR-80 spawns. It may spawn with a helicopter escort (this is randomized). If the convoy spawns with a helicopter escort, quit the mission and restart until it spawns without.

After the convoy spawns, you see a message



Followed quickly by a second message



Sit back and watch the convoy proceed. You may want to accelerate time (to, say 20x) until it reaches the road’s T-junction and turns east towards the two T-55.

When it reaches the T-junction, a message appears



Soon after the convoy heads east, it comes into range of the two T-55 and they start engaging the convoy:



The BTR-80a are no match for the T-55; a bit later



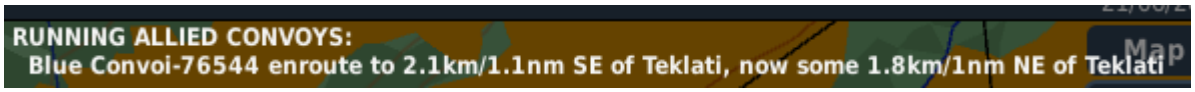
.

Followed by a message that there now a 0 blue convoys.

Now launch the same convoy again, this time *with* a helicopter escort. The two Cobras make short work of the two T-55. Yes, helicopter escorts can be a nasty piece of work, so be careful when you assign them in your mission as not to cut through too many defenses…

Since it is unopposed, this time the convoys makes it past all its checkpoints

While the convoy is moving, and close to Senaki Kolkhi, go to Communications→Other→Convoys and choose “List known Convoys”.



When it arrives at Senaki-Kolkhi, or more precisely, at the point 2.1 km SE of Teklati

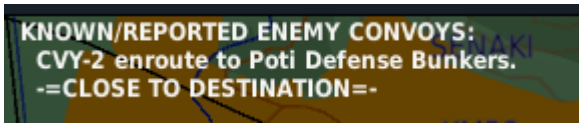


At that point in the mission, the convoy (vehicles and helicopters vanish)

Now, go to Communications→Other→Convoy Control→Spawn Convoys and choose “Start RED Coastal Convoy”. A RED convoy starts (consisting of either two or 5 vehicles), escorted by helicopter(s): either a Huey or two Cobras. If you are sill in the BLUE “Rubber Duck” Frogfoot, a message appears



Call up the “List known Convoys” report



Side note: Remember that “nasty piece of work” bit? If the red convoy that you spawned is escorted by Cobras and you sit on Senaki-Kolkhi’s tarmac in a blue Frogfoot, there is a good chance that one of the Cobras decides that it doesn’t like you and attempts to kill you.

After a (rather long) while, it arrives in Poti and disappears:



Lastly, go to Communications→Other→Locator and choose “Tell me where I am”:



8.87.2.2 ME

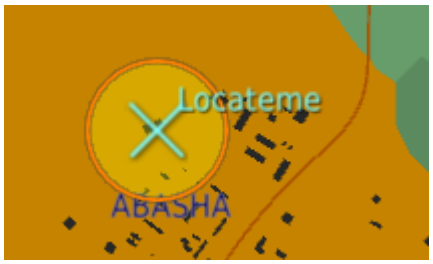
Named Map Locations

Let's begin with the easiest first, as it also serves well to describe what the twn module does behind the scenes: When the player invokes the "Tell me where I am command". This simply triggers the messenger in the "Locateme" zone:

Name	Value	
messenger?	where	
message	I am <loc:Locateme>.	
radioMenu	Locator	
itemA	Tell me where I am	
A!	where	

As you can see, it's a simple compound zone, stacking a radioMenu to trigger a flag where, and a messenger that triggers on the where flag.

When the messenger is triggered, it outputs the message "`I am <loc:Locateme>.`". The interesting bit here is the wildcard <loc:Locateme> which uses the twn module to translate a point into a map location. "Locateme" is a trigger zone, so the center of the trigger zone is used as reference for the location.



The trigger zone is north of a village called Abasha on the map. Twn retrieves the position and name of the closest map locale, measures the distance, and then creates an appropriate message:



Outside of wildcards, this ability of twn to retrieve map locations becomes automatically available to other modules, and all that mission designers must do to access it is to add the twn module to the mission.

Convoys

Let's look at red convoy that we spawned last, as it looks deceptively simple, yet packs a lot of cool features. Let us inspect this convoy's setup first:



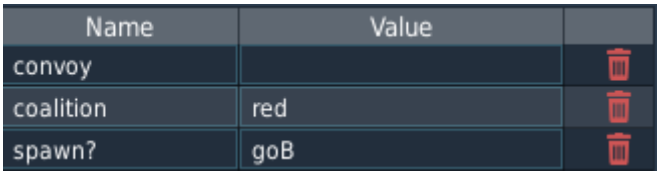
If you look closely, you'll see that there are two groups of ground vehicles inside this zone: a blue (three units) and a red (five units) group. This is to demonstrate multiple features:

a) that Convoy doesn't care what side a convoy belongs to when a group is defined. When the convoy is spawned, ownership is determined by the zone's *coalition* attribute (which is red)
b) You can place multiple groups inside a convoy zone (and only one unit has to actually be inside the trigger zone). When a convoy is launched, one of the groups is chosen by random

There are two helicopter groups inside the “Durgena Red Supply” trigger zone: Like with the ground vehicles, you can add multiple helicopter groups to a convoy zone, and when the convoy spawns, one of the possible escort groups is chosen at random.

The convoy zone itself holds no surprises:

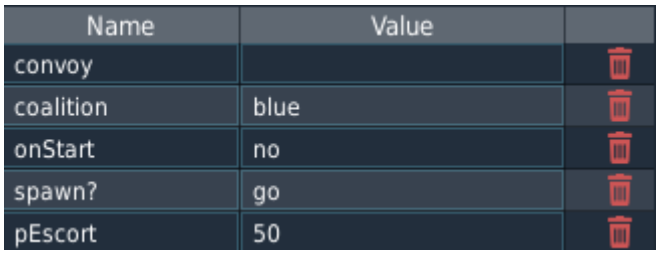
coalition is set to red (which makes all convoys started by this zone members of RED coalition), irrespective of which coalition the groups belonged to when the groups were set up in Mission Editor.



The convoy spawns when the flag *goB* that is connected to the *spawn?* input changes. All other attributes are using their default values, meaning that – since there are helicopter groups defined – there is always a helicopter escort (pEscort defaults to 100%) and that no convoy spawns at mission start.

Another perhaps interesting aspect: although one of the helicopter groups has a (simple) route defined, that is ignored and the helicopters escort the convoy, ignoring any previously defined routes.

If we now turn our attention to the blue convoy, we see that its setup is more simple: a single group of vehicles, and a single helicopter group. However, the convoy’s pEscort attribute is set to 50, meaning that only in 50% of all spawns the convoy should receive a helicopter escort. This is the reason that you may have had to try multiple times for the escort to start un-escorted. We also see here that the *onStart* attribute is set to false – something that isn’t necessary (since it defaults to false) but something I like to do for some zones to be more explicit should I later want to maintain the mission.



8.87.3 Discussion

On the Menu

There is a bit more depth to the convoy module that we want to explore in this section. Convoy comes with its own built-in GUI, and when enabled usually installs itself at the Communications→Other level. Here we use the config zone’s attachTo: attribute to attach Convoy’s convoy status menu to a Main Menu named “Convoy Main Menu” provided by radioMenu. That radio menu provides further convoy options, like launching the various convoys.



Counting blue Convoys
There's another attribute in Convoy's config: *blueConvoy#*. This is a direct output that connects to the flag named *blueCNum* and always is updated to the number of currently running blue convoys. Mission designers can use to trigger many effects, here we merely use it to trigger a messenger that always updates us when the number of running blue convoys changes:

Name	Value	
messenger?	blueCNum	
message	Number of blue convoys : <	

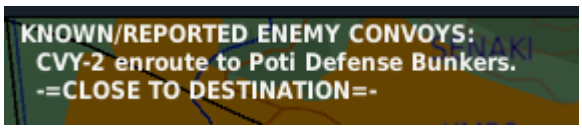
Again, few surprises: the messenger triggers every time that the value of the flag blueCNum changes. Since blueCNum is fed directly from Convoy's 'living blue convoy' counter, this messenger trigger whenever a new blue convoy is created or a blue convoy is destroyed (note that this can lead to an unlikely but still possible side effect: if within a short time a blue convoy is created and destroyed, the net result on the direct output may not change, and the messenger nit trigger. This unlikely even becomes very likely indeed if you feed a convoy's "*dead!*" output into its "*spawn?*" input to create ever-running convoys. A side-effect that you need to be aware of if you use Convoy's direct outputs to monitor the flow of convoys in your mission)

Challenges:

Where am I?
Let's start simple. The messenger that tells you where I am is set up inefficiently: it hogs the flag named 'where'. Change this so that, as a true stacked module, it uses a local flag.

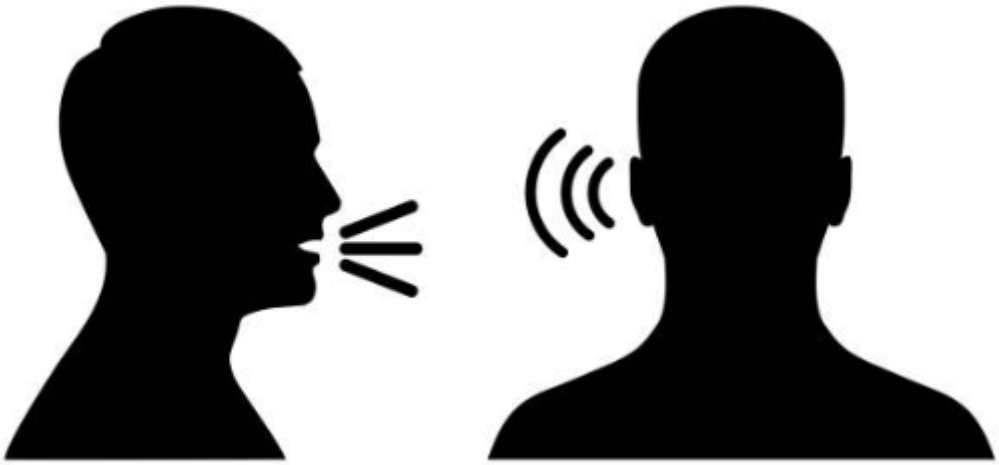
And now let's make this stack more useful: use it to always tell us the location of the unit "Rubber Duck".

So close!
When you start the red convoy and request a report, it always screams at everyone on BLUE that it is CLOSE TO DESTINATION!!!!one



Find out why, and change the set up so that this warning only comes up when the convoy is within 2 miles of Poti. Remember that there are two possible groups of convoys.

8.88 The Yeller Model



8.88.1 Demonstration Goals

This miz functions as a hands-on example for the “Yeller Model” explanation introduced early in this doc: it’s a simplified, yet highly effective way to understand how how modules talk to each other, and it completely avoids “Flags”. Please read the relevant chapter, as it (hopefully) makes understanding a central aspect of DML much more accessible.

8.88.2 What To Explore

8.88.2.1 In Mission

Start the mission and enter the Frogfoot.

A message “Time to go now” appears on-screen.



Wait.

Every 90 seconds, another message “Time to go” appears.

Get bored, end mission.

8.88.2.2 ME

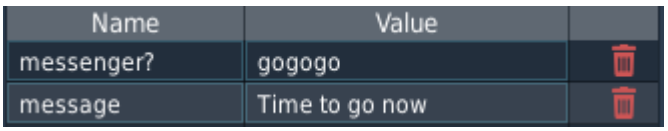
8.88.3 Discussion

As per “Yeller Model”, we use different modules: some yell commands, others listen for commands, some to both. In this mission, we are using the Yeller module pulseZone, which sends the “gogogo” command (or “yells gogogo”) over its “pulse!” output (in classic “Flag Model”, the module changes the value of the Flag named “gogogo”).

Name	Value	
pulse!	gogogo	
time	90	

As we can see, the pulser is set up to send the “gogogo” command every 90 seconds (time = 90) on the pulse! output.

The Listener module is a messenger that waits for the command “gogogo”. When it hears the command, it outputs the message “Time to go now” to the screen.



As you can see, using the “Yeller Model” can dramatically help to understand how the various modules talk to each other without losing significant context. It’s up to you to take the simplified approach (“commands” versus “signals on flags”) – as long as it works for you.

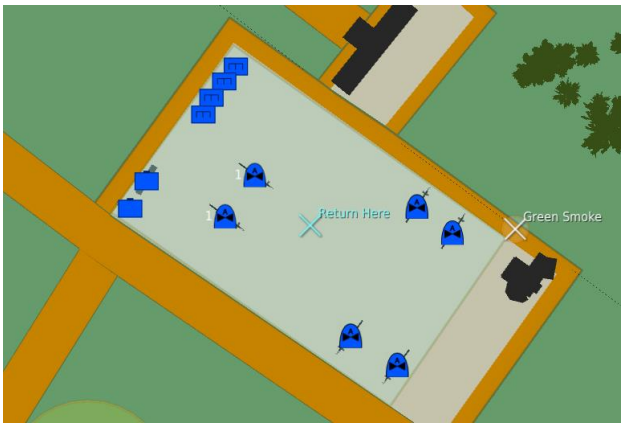
8.89 Inferno at Sea (heloTroops: dropZone)

8.89.1 Demonstration Goals

HeloTroops, due to its ability to allow players to pick up and transports infantry from anywhere to everywhere on the map, has proven to be one of the most successful and fun modules. Even though it was conceived as a drop-in module, mission designers can use additional built-in functionality to fine-tune its operation. This mission high-lights heloTroop's dropZones – it's mechanism to restrict troop delivery to certain places, and send signals (commands) when players deliver units to

(IMG)

In this demo we use a dropZone to restrict all troop deployment from helicopters to the "Return Here" zone on Batumi's Tarmac close to the tower:



8.89.2 What To Explore

8.89.2.1 In Mission

Note: This mission requires that you fly a helicopter well enough to land on an off-shore oil platform, and return that helicopter back to Batumi and land inside a defined area,

Note This mission requires a troop-transport capable helicopter: Huey, Hip, Hind or Hook. The mission uses a Huey, you can simply add one of the other helicopters and the mission will work

Start in the Huey and fly to the burning platform west of Batumi. Land on the Platform.

The three infantry units board the helicopter automatically

Take off and return to Batumi, and land on the runway. Note that no units disembark.

Now land on the tarmac next to the Tower (look for 4 Hueys and firetrucks) where you initially started.

As soon as you touch down, the infantry disembarks, and you see a message "Congratulations – You have saved 1 groups of evacuees" (or similar).

60 seconds later, the units that disembarked your helicopter disappear.

Note that at no point in time was there any player interaction required for the units to hop on or off the helicopter.

8.89.2.2 ME

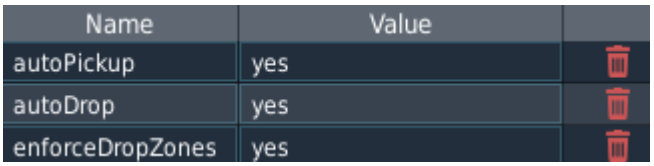
This simple demo entirely relies on heloTroops' dropZone abilities:there is a dropZone at Batumi's tarmac with two attributes:



The dropZone! Attribute itself that sends a signal on the flag *saved* connected to the output dropZone! ("yells the command *saved*") whenever an eligible player helicopter lands inside and disembarks troops.

The disembarked troops are automatically de-spawned after 60 seconds (*autoDespawn* = 60)

The mission's behavior is rounded off by some settings inside the configZone:



- most importantly, *enforceDropZones* is set to true, which prevents troops to disembark from player helicopters when the helicopter in on the ground. We saw this when we touched down on Batumi's runway and tried to unload our troops via F-10 Other.
- We also set troop pick-up and drop-off to automatic. This is mere mission polishing, as users can override this setting with heloTroops natice UI. This also demonstrates nicely that players cannot override the *enforceDropZones* setting manually.

8.89.3 Discussion / Challenges

There are some hidden niceties in this mission:

- The spawner on the oil platform spawns troops a minute after a helicopter has picked up the last batch. This is standard spawner behavior and works around a possible issue if you set the spawner to only spawn when the last batch was delivered to Batumi, and the helicopter crashed with units on board, or accidentally killed all units on the platform. This also allows multiplayer gameplay.
- The messenger counts the number of deliveries
- The fires have been put at an altitude to make them appear to come from within the platform

Challenges:

- Change the spawners setup so that units do not spawn automatically (i.e. set the spawner's paused attribute to true). Then figure out a safe way how you can spawn evacuees on the platform so that the mission is multiplayer-capable and can handle the case when a helicopter loaded with troops crashes and is lost.
- Have the messenger count the number of units saved instead of the number of groups.

8.90 Mission name

8.90.1 Demonstration Goals

8.90.2 What To Explore

8.90.2.1 In Mission

8.90.2.2 ME

8.90.3 Discussion

9 DML FAQ

Got a question? Check here first!

9.1 General

9.1.1 Loading Modules

- *How do I add modules?*
  There's a video available, and it's also explained in the chapter "Using DML" in this manual. In short, you add a 'Trigger Rule' to your mission "AT START", and then paste some text into an ACTION's text box.

- *Do I have to load DML at mission start?*
  That is the best point in time to do so. Modules assume that they load at mission start, and the information that they gather may not correctly reflect the current strategic situation if you do not.

- *Is the order in which I add modules relevant?*
  Yes, add 'dependent' modules before you add that module. For example, you must load *dcsCommon* and *cfxZones* before you load *messenger*. You **must** add the '*persistence'* module before any other module that should be able to persist data (i.e. my recommendation is that you load persistence directly after dcsCommon and cfxZones).

9.1.2 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 to 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'

9.2 Module Specific

9.2.1 CloneZone

- *The group that the cloner should spawn doesn't spawn*,
  even though I bang on the correct flag. When set to verbose it even claims to have spawned the groups, but nothing appears on the map.
  Check that you did **not check** the **Late Activation** flag in the group, as this too is copied to the clone, and the clone then forever waits to be activated.

- *Nothing spawns when the mission starts*
  Cloners default to removing all units in their zone (moving them to a template) and then do nothing. If you want a cloner to automatically clone when the mission starts, you must add an attribute ‘onStart’ and set its value to ‘true’

9.2.2 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.

9.2.3 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

9.2.4 PlayerScore

- *Units do not score correctly even though they are in the Score Table*
  Since DCS 2.7, a weird quirk in the game engine can turn units that are ‘cooking off’ (burning before they explode) into static objects. In the process they lose their connection to their group. As a result, PlayerScore cannot connect that unit to the group score. So if you award score by group name, units of that group can be affected by this. If this happens, PlayerScore awards the default score for that category instead.

- *I have set up score for scenery objects, but suddenly it is no longer awarded*
  This has the same root cause as the Object Destruct Detector issue: ED can change object ID for scenery objects from release to release. If you use scenery objects as scorable targets, you will have to check the IDs in you mission after each release.

9.2.5 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. After you install SSB you MUST restart DCS because all server files are onle read when DCS starts up

- *Did all that, still no joy*
  Is the mission running? Many servers are configured so that a multiplayer mission is paused when it begins. Blocking slots only happens once the mission is started.

9.2.6 stopGap/noGap

- *When players enter their cockpit, the plane falls to the ground, damaging it*
  Your mission is experiencing synchronization issues, a common DCS bug. Install ‘stopGapGUI’ or ‘noGapGUI’ on your server to work around this bug in DCS server.