MIssion Scripting Tools (Mist) Version 4.2 Guide

Guide rev. 0

Authors: Grimes, Speed

Table of Contents

Introduction

MIssion **S**cripting **T**ools (Mist) is a collection of Lua functions and databases that is intended to be a supplement to the standard Lua functions included in the simulator scripting engine. Mist functions and databases provide ready-made solutions to many common scripting tasks and challenges, enabling easier scripting and saving mission scripters time. The table mist.flagFuncs contains a set of Lua functions (that are similar to Slmod functions) that <u>do not</u> require detailed Lua knowledge to use. However, the majority of Mist does require knowledge of the Lua language, and, if you are going to utilize these components of Mist, it is necessary that you read the Simulator Scripting Engine guide on the official ED wiki.

Online Documentation and Links

The MIST online documentation can be found on the hoggit wiki. Both the wiki and this PDF will still be updated with each mist release.

http://en.wiki.eagle.ru/wiki/DCS_Mission_Editor_(ME) (currently out of date and unable to edit)

http://wiki.hoggit.us/view/Simulator_Scripting_Engine_Documentation

http://wiki.hoggit.us/view/Mission_Scripting_Tools_Documentation

Forum Links

http://forums.eagle.ru/showpost.php?p=1622305&postcount=3

https://github.com/mrSkortch/MissionScriptingTools

How to Load It

<u>Mist is not a mod of the game files.</u> You include it into your missions simply by using the DO SCRIPT FILE trigger actions to load Mist at or near mission start. The screenshot below illustrates trigger logic that loads Mist at mission time = 1 sec (using DO SCRIPT FILE).

(Also note that in the above screenshot the trigger named "Mission Script". This trigger is similar to the trigger activating mist, however the condition is "Time More (2)." The mission script used in this example requires Mist to be loaded first, otherwise a LUA error will occur and the mission will not function correctly.)

Once your mission trigger logic loads Mist, Mist exists within the global mission scripting Lua environment, and its components are accessible in all subsequent scripts run with DO SCRIPT, DO SCRIPT FILE, AI Lua stop conditions, etc. The 1.2.4 patch added the "initialization" script or script file option in the triggers list.

NOTE #1:

Due to its size, mist must be loaded via a DO SCRIPT FILE trigger action. Mist is simply to large to paste into a do script box.

NOTE #2:

It is best if Mist is loaded into your mission as early as possible. Unlike Slmod, Mist should not be vulnerable to being used too early, and it will be fine if you load it on a MISSION START type trigger.

Function Definition Format in this Guide

Currently, this guide uses the following format for function definitions:

ReturnedValueType, … **functionName(RequiredVariableType** *RequiredVariableName***, … ,**
OptionalVariableType *OptionalVariableName***, …)**

So, for example:

table **mist.getUnitsInZones (UnitNameTable** *unit_names***, table** *zone_names***, string**
*zone_type***)**

This function accepts a required variable/value of type UnitNameTable, followed by a second
required variable of type table, followed by an optional variable of type string. It returns a
single value of type table.

UnitNameTables

Many Mist functions require tables of unit names, which are known in Mist as *UnitNameTables*.
These follow a special set of shortcuts borrowed from Slmod. These shortcuts alleviate the
problem of entering huge lists of unit names by hand, and in many cases, they remove the need
to even *know* the names of the units in the first place!

These are the unit table "short-cut" commands (there will be examples of how to use them in a
moment):

Character sequence + name commands:
"[-u]<unit name>" - subtract this unit from the table
"[g]<group name>" - add this group's units to the table
"[-g]<group name>" - subtract this group's units from the table
"[c]<country name>" - add this country's units to the table
"[-c]<country name>" - subtract this country's units from the table

Stand-alone identifiers
"[all]" – add all units to the table
"[-all]" – remove all units from the table
"[blue]" - add all blue coalition units to the table
"[-blue]" - subtract all blue coalition units from the table
"[red]" - add all red coalition units to the table
"[-red]" - subtract all red coalition units from the table

Compound identifiers:
"[c][helicopter]<country name>" - add all of this country's helicopters to the table

"[-c][helicopter]<country name>" - subtract all of this country's helicopters from the table
"[c][plane]<country name>" - add all of this country's planes to the table
"[-c][plane]<country name>" - subtract all of this country's planes from the table
"[c][ship]<country name>" - add all of this country's ships to the table
"[-c][ship]<country name>" - subtract all of this country's ships from the table
"[c][vehicle]<country name>" - add all of this country's vehicles to the table
"[-c][vehicle]<country name>" - subtract all of this country's vehicles from the table

"[all][helicopter]" - add all helicopters to the table
"[-all][helicopter]" - subtract all helicopters from the table
"[all][plane]" - add all planes to the table
"[-all][plane]" - subtract all planes from the table
"[all][ship]" - add all ships to the table
"[-all][ship]" - subtract all ships from the table
"[all][vehicle]" - add all vehicles to the table
"[-all][vehicle]" - subtract all vehicles from the table

"[blue][helicopter]" - add all blue coalition helicopters to the table
"[-blue][helicopter]" - subtract all blue coalition helicopters from the table
"[blue][plane]" - add all blue coalition planes to the table
"[-blue][plane]" - subtract all blue coalition planes from the table
"[blue][ship]" - add all blue coalition ships to the table
"[-blue][ship]" - subtract all blue coalition ships from the table
"[blue][vehicle]" - add all blue coalition vehicles to the table
"[-blue][vehicle]" - subtract all blue coalition vehicles from the table

"[red][helicopter]" - add all red coalition helicopters to the table
"[-red][helicopter]" - subtract all red coalition helicopters from the table
"[red][plane]" - add all red coalition planes to the table
"[-red][plane]" - subtract all red coalition planes from the table
"[red][ship]" - add all red coalition ships to the table
"[-red][ship]" - subtract all red coalition ships from the table
"[red][vehicle]" - add all red coalition vehicles to the table
"[-red][vehicle]" - subtract all red coalition vehicles from the table

Country names for the "[c]<country name>" and "[-c]<country name>" short-cuts:

Turkey
Norway
The Netherlands
Spain
UK
Denmark

USA
Georgia
Germany
Belgium
Canada
France
Israel
Ukraine
Russia
South Ossetia
Abkhazia
Italy
Australia
Austria
Belarus
Bulgaria
Czech Republic
China
Croatia
Finland
Greece
Hungary
India
Iran
Iraq
Japan
Kazakhstan
North Korea
Pakistan
Poland
Romania
Saudi Arabia
Serbia, Slovakia
South Korea
Sweden
Switzerland
Syria
USAF Aggressors

Do NOT use a '[u]' notation for single units. Single units are referenced the same way as before: simply input their names as strings.

These unit tables are evaluated in order, and you cannot subtract a unit from a table before it is added. For example,

```
{'[blue]', '[-c]Georgia'}
```

will evaluate to all of blue coalition except those units owned by the country named "Georgia"; however:

```
{'[-c]Georgia', '[blue]'}
```

will evaluate to all of the units in blue coalition, because the addition of all units owned by blue coalition occurred AFTER the subtraction of all units owned by Georgia (which actually subtracted nothing at all, since there were no units in the table when the subtraction occurred).

More examples:

```
{'[blue][plane]', '[-c]Georgia', '[-g]Hawg 1'}
```
-Evaluates to all blue planes, except those blue units owned by the country named "Georgia" and the units in the group named "Hawg1".

```
{'[g]arty1', '[g]arty2', '[-u]arty1_AD', '[-u]arty2_AD', 'Shark 11' }
```
-Evaluates to the unit named "Shark 11", plus all the units in groups named "arty1" and "arty2" except those that are named "arty1_AD" and "arty2_AD".

If you want to write your own scripts that make use of the Mist function that creates UnitNameTables, then this is the function you use:

mist.makeUnitTable
table   mist.makeUnitTable (table *t*)

This function accepts table *t* (which must be indexed sequentially starting at 1), applies the UnitNameTable short-cut rules, and returns a new table of unit names. Also, this function adds the table key and value
```
processed = time
```
to the returned table, indicating that the table has already been "run through" the UnitNameTable processing/shortcuts (useful for self-rescheduling functions- obviously, you only need to apply the UnitNameTable rules the first time a self-rescheduling function is run!). The time value returned in the processed entry is the time at which the function was run and the list was populated.

Flag Functions

The mist "Flag functions" are functions that are similar to Slmod functions that detect a game condition and set a flag when that game condition is met. They are intended to be used by persons with little or no experience in Lua programming, but with a good knowledge of the DCS mission editor.

mist.flagFunc.mapobjs_dead_zones

mist.flagFunc.mapobjs_dead_zones(table *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{
zones = **table** *zones*,
flag = **number/string** *flag*,
stopflag = **number/string** *stopflag*,
req_num = **number** *req_num*
}

Once this function is run, it will start a continuously evaluated process that will set flag *flag* true if map objects (such as bridges, buildings in town, etc.) die (or have died) in a mission editor zone (or set of zones). This will only happen once; once the flag *flag* is set true, the process ends.

zones is a table of zone names (indexed numerically).

stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

req_num is an optional variable that allows you to specify the minimum number of map objects that have die for *flag* to be set true. If *req_num* is not specified, it defaults to 1.

As of right now, this function detects all map objects that have EVER died, even ones that died before the function ran (unlike the similar slmod.mapobjs_dead_in_zone function). This could probably be improved upon in future releases of Mist.

Examples:
```
mist.flagFunc.mapobjs_dead_zones{ zones = {'bridge1'}, flag = 51 }
--[[Once run, this function will set flag 51 is set true if/when a map object
has died/dies within the zone named "bridge1".]]
```

```
mist.flagFunc.mapobjs_dead_zones{
    zones = {'town1', 'town2', 'town3'},
    flag = 1050,
    req_num = 5
}
--[[Once run, this function will set flag 1050 true if/when 5 or more
map objects have died/die within the zones named "town1", "town2", and
"town3".]]


mist.flagFunc.mapobjs_dead_zones{
    zones = {'Vaziani', 'Sogunlug', 'tbilisi-lochini'},
    flag = 999,
    req_num = 10,
    stopflag = 10000
}
--[[Once run, this function will set flag 999 true if/when 10 or more
map objects have died/die within the zones named "Vaziani", "Sogunlug", and
"tbilisi-lochini"- UNLESS flag 10000 (the stopflag) becomes true first!]]
```

mist.flagFunc.mapobjs_dead_polygon

mist.flagFunc.mapobjs_dead_polygon(table *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{
zone = **table** *zone,*
flag = **number/string** *flag,*
stopflag = **number/string** *stopflag*,
req_num = **number** *req_num*
}

Once this function is run, it will start a continuously evaluated process that will set flag *flag* true once map objects (such as bridges, buildings in town, etc.) die (or have died) within a polygon-shaped zone. This will only happen once; once the flag *flag* is set true, the process ends.

zone is a table of map points that defines the polygon shape (indexed numerically). See mist.pointInPolygon for a good explanation.

 stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

req_num is an optional variable that allows you to specify the minimum number of map objects that have die for *flag* to be set true. If *req_num* is not specified, it defaults to 1.

As of right now, this function detects all map objects that have EVER died, even ones that died before the function ran. This could probably be improved upon in future releases of Mist.

Examples:

```
mist.flagFunc.mapobjs_dead_polygon{
    zone = mist.getGroupPoints('Russia group'),
    flag = 90,
    req_num = 50,
}
--[[Once run, this function will set flag 90 true if/when 50 or more
map objects have died/die within the polygon shape defined by the waypoints of
the group named "Russia group".]]


mist.flagFunc.mapobjs_dead_polygon{
    zone = {
        [1] = mist.DBs.unitsByName['NE corner'].point,
        [2] = mist.DBs.unitsByName['SE corner'].point,
        [3] = mist.DBs.unitsByName['SW corner'].point,
        [4] = mist.DBs.unitsByName['NW corner'].point
    },
    flag = 151,
    req_num = 15,
}
--[[Once run, this function will set flag 151 true if/when 15 or more
map objects have died/die within the polygon shape defined by the initial
starting positions of the units named "NE corner", "SE corner", "SW corner",
and "NW corner".]]
```

mist.flagFunc.units_in_zones
mist.flagFunc.units_in_zones(table *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **UnitNameTable** *units,*
zones = **table** *zones,*
flag = **number/string** *flag,*
stopflag = **number/string** *stopflag,*
zone_type = **string** *zone_type,*
req_num = **number** *req_num,*
interval = **number** *interval,*
toggle = **boolean** *toggle,*
unitTableDef = **table** *unitTableDef,*

}

Once this function is run, <u>it will start a continuously evaluated process</u> that will set flag *flag* true once units from *units* are inside any one of a series of zones listed in *zones*. This process will keep running, and *flag* will keep being set true as long as the unit(s)-in-zone(s) conditions persist, unless the process is stopped with *stopflag*.

Important note: With mist v4 this function has an added variable to account for any dynamically added units to the mission. If you use a unitNameTable definition such as (`units = {'[blue] [vehicle]'})` then the definition would be saved and the list can be updated if units are dynamically added to the mission. If this function is directly given a list of specific units such as (`units = {'Chevy11', 'Chevy12', 'Chevy13', 'Chevy14'})` then the function will only work on those specific units.

units is a UnitNameTable- a table of unit names that follow a special set of rules (see the entry on UnitNameTables).

zones is a table of zone names (indexed numerically).

stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

req_num is an optional variable that allows you to specify the minimum number of units that must be in one (or more) of the zones before *flag* is set true. <u>If *req_num* is not specified, it defaults to 1.</u>

zone_type is an optional variable that defines the shape of the zones. The following are the allowed values for *zone_type*:

> `'cylinder'` – cylindrical shaped zone extending to +/- infinity in altitude.
> `'sphere'` – spherical zone.
>
> If not specified, it defaults to `'cylinder'`.

interval is an optional variable that allows you to specify how often (in seconds) the in-zone condition is checked; for lots of units in lots of zones (like hundreds of units in hundreds of different zones), it might be desirable to increase the interval to save computer processing time. <u>If not specified, interval defaults to 1.</u>

toggle is an optional variable that if present will switch the *flag* value to false when the required conditions are not met. If not specified toggle defaults to false.

unitTableDef is an optional variable that is used to define a unitTableName definition. This variable is automatically populated by whatever is passed in the units table if this entry is not passed.

Examples:

```
mist.flagFunc.units_in_zones{
    units = {'Chevy11', 'Chevy12', 'Chevy13', 'Chevy14'},
    zones = {'Mozdok', 'Krymsk', 'Anapa', 'Mineral'},
    flag = 100,
    zone_type = 'sphere'
}
--[[Once run, this function will start a process that will set flag 100
true when any of the units named "Chevy11", "Chevy12", "Chevy13", or
"Chevy14" are in any of the spherical-shaped zones named "Mozdok",
"Krymsk", "Anapa", or "Mineral".]]


mist.flagFunc.units_in_zones{
    units = {'[blue][vehicle]'},
    zones = {'Point Bastion'},
    flag = 99,
    req_num = 8,
    stopflag = 1000
}
--[[Once run, this function will start a process that will set flag 99 true
8 or more blue vehicles are within the zone named "Point Bastion", UNLESS
flag 1000 becomes true first.]]
```

mist.flagFunc.units_in_moving_zones

mist.flagFunc.units_in_moving_zones(table *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **UnitNameTable** *units,*

zone_units = **UnitNameTable** *zone_units,*

flag = **number/string** *flag,*

radius = **number** *radius,*

stopflag = **number/string** *stopflag,*

zone_type = **string** *zone_type,*

req_num = **number** *req_num,*

interval = **number** *interval,*

toggle = **boolean** *toggle,*
unitTableDef = **table** *unitTableDef,*
}

Important note: With mist v4 this function has an added variable to account for any dynamically added units to the mission. If you use a unitNameTable definition such as (`units = {'[blue]`
`[vehicle]'})` then the definition would be saved and the list can be updated if units are dynamically added to the mission. If this function is directly given a list of specific units such as (`units = {'Chevy11', 'Chevy12', 'Chevy13', 'Chevy14'})` then the function will only work on those specific units.

Once this function is run, <u>it will start a continuously evaluated process</u> that will set flag *flag* true once units listed in *units* are inside any of the moving zones around the units listed in *zone_units*. This process will keep running, and *flag* will keep being set true as long as the unit(s)-in-zone(s) conditions persist, unless the process is stopped with *stopflag*.

units is a UnitNameTable- a table of unit names that follow a special set of rules (see the entry on UnitNameTables).

zone_units is a UnitNameTable- a table of unit names that follow a special set of rules (see the entry on UnitNameTables).

radius – the radius, in meters, of all the zones drawn around each unit in *zone_units*.

stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

req_num is an optional variable that allows you to specify the minimum number of units that must be in one (or more) of the moving zones before *flag* is set true. If *req_num* <u>is not specified, it defaults to 1.</u>

zone_type is an optional variable that defines the shape of the moving zone drawn around each zone unit. The following are the allowed values for *zone_type*:

       `'cylinder'` – cylindrical shaped zone extending to +/- infinity in altitude.
       `'sphere'` – spherical zone.

       If not specified, it defaults to `'cylinder'`.

interval is an optional variable that allows you to specify how often (in seconds) the in-zone condition is checked; for lots of units and zone units (like hundreds of units and hundreds of zone units), it might be desirable to increase the interval to save computer processing time. <u>If not specified, interval defaults to 1.</u>

toggle is an optional variable that if present will switch the *flag* value to false when the required conditions are not met. If not specified toggle defaults to false.

unitTableDef is an optional variable that is used to define a unitTableName definition. This variable is automatically populated by whatever is passed in the units table if this entry is not passed.

<u>Examples:</u>

```
mist.flagFunc.units_in_moving_zones{
    units = {'[g]M1 PLT1', '[g]M1 PLT2', '[g]M1 PLT3', '[g]M1 PLT4' },
    zone_units = {'[red][vehicle]'},
    flag = 51,
    radius = 6500,
}
--[[Once run, this function will start a process that will set flag 51
true when any of the units in the groups named "M1 PLT1", "M1 PLT2",
"M1 PLT3", or "M1 PLT4" is within 6500 meters of any red vehicle.]]


mist.flagFunc.units_in_moving_zones{
    units = {'[blue]'},
    zone_units = {'[red]'},
    flag = 500,
    radius = 10000,
    stopflag = 9999,
    req_num = 4,
    zone_type = 'sphere',
    interval = 10,
}
--[[Once run, this function will start a process that will set flag 500
true when at least 4 blue units are within 10000 meters of any blue
units.  The process will run once every 10 seconds unless flag 9999
becomes true.]]
```

mist.flagFunc.units_in_polygon

mist.flagFunc.units_in_polygon(table *vars*)

<u>vars has the following recognized fields (required entries in blue, optional in green):</u>

```
{
units = UnitNameTable units,
zone = table zone,
flag = number/string flag,
stopflag = number/string stopflag,
maxalt = number maxalt,
req_num = number req_num,
interval = number interval,
toggle = boolean toggle,
unitTableDef = table unitTableDef,

}
```

Once this function is run, it will start a continuously evaluated process that will set flag *flag* true once units listed in *units* are inside the polygon zone defined by the map points listed in *zone*. This process will keep running, and *flag* will keep being set true as long as the unit(s)-in-zone conditions persist, unless the process is stopped with *stopflag*.

Important note: With mist v4 this function has an added variable to account for any dynamically added units to the mission. If you use a unitNameTable definition such as (`units = {'[blue] [vehicle]'})` then the definition would be saved and the list can be updated if units are dynamically added to the mission. If this function is directly given a list of specific units such as (`units = {'Chevy11', 'Chevy12', 'Chevy13', 'Chevy14'})` then the function will only work on those specific units.

units is a UnitNameTable- a table of unit names that follow a special set of rules (see the entry on UnitNameTables).

zone is a table of map points that defines the polygon shape (indexed numerically). See mist.pointInPolygon for a good explanation.

 stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

req_num is an optional variable that allows you to specify the minimum number of units that must be in the polygon zone before *flag* is set true. If *req_num* is not specified, it defaults to 1.

maxalt is an optional variable that allows you to a specify a maximum altitude (above sea level) for the polygon zone. Altitude above ground level will likely be added in a future version of Mist.

interval is an optional variable that allows you to specify how often (in seconds) the in-zone condition is checked; perhaps if you are checking thousands of units it might be useful to use this variable to reduce computation time. This could also be useful if you needed the flag *flag* to be set true less often than once per second.

toggle is an optional variable that if present will switch the *flag* value to false when the required conditions are not met. If not specified toggle defaults to false.

unitTableDef is an optional variable that is used to define a unitTableName definition. This variable is automatically populated by whatever is passed in the units table if this entry is not passed.

Examples:

```
mist.flagFunc.units_in_polygon{
    units = {'[blue][vehicle]'},
    zone = mist.getGroupPoints('forest1'),
    flag = 11
}
--[[Once run, this function will start a process that will set flag 11
true when any blue vehicles are within the polygon shape created by the
waypoints of the group named "forest1"]]


mist.flagFunc.units_in_polygon{
    units = {'[red][plane]'},
    zone = {
        [1] = mist.DBs.unitsByName['AO 1'].point,
        [2] = mist.DBs.unitsByName['AO 2'].point,
        [3] = mist.DBs.unitsByName['AO 3'].point,
        [4] = mist.DBs.unitsByName['AO 4'].point,
        [5] = mist.DBs.unitsByName['AO 5'].point,
        [6] = mist.DBs.unitsByName['AO 6'].point,
    },
    flag = 201,
    maxalt = 6000,
    interval = 30
}
--[[Once run, this function will start a process that will set flag 201
true when any red planes are within the polygon shape derived by the
intial starting positions of the units named "AO 1" through "AO 6" and
are less than 6000 meters above sea level.  This process will run once
every 30 seconds.]]
```

mist.flagFunc.units_LOS
mist.flagFunc.units_LOS(table *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
unitset1 = **UnitNameTable** *unitset1,*
altoffset1 = **number** *altoffset1,*
unitset2 = **UnitNameTable** *unitset2,*
altoffset2 = **number** *altoffset2,*
flag = **numbe/stringr** *flag,*
stopflag = **number/string** *stopflag,*
radius = **number** *radius,*
req_num = **number** *req_num,*
interval = **number** *interval,*
toggle = **boolean** *toggle,*
unitTableDef1 = **table** *unitTableDef1,*
unitTableDef2 = **table** *unitTableDef2,*
}

Important note: With mist v4 this function has an added variable to account for any dynamically added units to the mission. If you use a unitNameTable definition such as (`units = {'[blue] [vehicle]'})` then the definition would be saved and the list can be updated if units are dynamically added to the mission. If this function is directly given a list of specific units such as (`units = {'Chevy11', 'Chevy12', 'Chevy13', 'Chevy14'})` then the function will only work on those specific units.

Once this function is run, it will start a continuously evaluated process that will set flag *flag* true once units listed in *unitset1* are line-of-sight (LOS) to units listed in *unitset2*. This process will keep running, and *flag* will keep being set true as long as the line-of-sight conditions persist, unless the process is stopped with *stopflag*.

unitset1 and *unitset2* are UnitNameTables- a tables of unit names that follow a special set of rules (see the entry on UnitNameTables).

altoffset1 and *altoffset2* are the number of meters above the position of each unit in *unitset1* and *unitset2* (respectively) that the LOS sighting takes place from. So for example, for tanks,

you might want to use an altoffset value of 3. For airplanes, you would probably want to use an altoffset value of 0 (unless it was like an AWACs and you wanted the sighting point to be above it to simulate the AWACs radar dish); for a really tall search radar, perhaps you would want to use an altoffset value of like 12.

 stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

req_num is an optional variable that allows you to specify the minimum number of units that must be LOS before *flag* is set true. If *req_num* is not specified, it defaults to 1.

radius is an optional variable that allows you to a specify a maximum radius out to which the LOS check occurs; beyond this radius, units will not be considered LOS.

interval is an optional variable that allows you to specify how often (in seconds) the LOS conditions are checked; this could be used for a variety of reasons including saving computational time, or making *flag* be set true less often than once per second.

unitTableDef1 and *unitTableDef2* is an optional variable that is used to define a unitTableName definition. This variable is automatically populated by whatever is passed in the units table if this entry is not passed.

Examples:

```
mist.flagFunc.units_LOS{
    unitset1 = {'101'},
    altoffset1 = 0,
    unitset2 = {'AWACs'},
    altoffset2 = 6.5,
    flag = 101,
    interval = 7
}
--[[Once run, this function will start a process that will set flag 101
true when the unit named "101" is line of sight to the unit named
"AWACs".  This check will occur once every 7 seconds.]]

mist.flagFunc.units_LOS{
    unitset1 = {'[g]M1 PLT1', '[g]M1 PLT2', '[g]M2 PLT1'},
    altoffset1 = 3,
    unitset2 = {'[red][vehicle]'},
    altoffset2 = 3,
    flag = 10,
    interval = 20
}
```

```
--[[Once run, this function will start a process that will set flag 10
true when any unit in the groups named "M1 PLT1", "M1 PLT2", or "M2 PLT1"
are line of sight to any red vehicles.  This will occur once every 20
seconds.]]
```

mist.flagFunc.group_alive

mist.flagFunc.group_alive (table *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
groupName = **string** *groupName,*
flag = **number/string** *flag,*
stopflag = **number/string** *stopflag,*
interval = **number** *interval,*
toggle = **boolean** *toggle,*
}

Once this function is run, it will start a continuously evaluated process that will set flag *flag* true if the specified group is alive. If the *toggle* variable is present the flag will be set to false if the group is dead. This process is stopped with *stopflag*.

groupName is the name of the group which the function checks if it is alive or not.

flag is a number corresponding to the flag that will set true if the group is alive

 stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

interval is an optional variable that allows you to specify how often (in seconds) the group_alive condition is checked. If not specified, interval defaults to 1.

toggle is an optional variable that if present will switch the *flag* value to false when the required conditions are not met. If not specified toggle defaults to false.

Examples:

```
mist.flagFunc.group_alive {
    units = 'myGroup,
    flag = 100,
    toggle = true,
```

```
}
```

```
Mission Editor Trigger: Switched Condition> Flag 100 is True> Message to All
(Dynamically Spawned group is alive)
```

mist.flagFunc.group_dead

mist.flagFunc.group_dead (table *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
groupName = **string** *groupName,*
flag = **number/string** *flag,*
stopflag = **number/string** *stopflag,*
interval = **number** *interval,*
toggle = **boolean** *toggle,*
}

Once this function is run, <u>it will start a continuously evaluated process</u> that will set flag *flag* true if the specified group is dead. If the *toggle* variable is present the flag will be set to false if the group is dead. This process is stopped with *stopflag*.

groupName is the name of the group which the function checks if it is dead or not.

flag is a number corresponding to the flag that will set true if the group is dead

stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

interval is an optional variable that allows you to specify how often (in seconds) the group_dead condition is checked. <u>If not specified, interval defaults to 1.</u>

toggle is an optional variable that if present will switch the *flag* value to false when the required conditions are not met. If not specified toggle defaults to false.

Examples:

```
mist.flagFunc.group_dead {
    units = 'myGroup,
    flag = 100,
    toggle = true,
```

```
}
```

Mission Editor Trigger: Switched Condition> Flag 100 is True> Do
Script(mist.respawnGroup('myGroup'))

mist.flagFunc.group_alive_less_than
mist.flagFunc.group_alive_less_than (table *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{
groupName = **string** *groupName*,
flag = **number/string** *flag*,
percent = **number** *percent*,
stopflag = **number/string** *stopflag*,
interval = **number** *interval*,
toggle = **boolean** *toggle*,
}

Once this function is run, it will start a continuously evaluated process that will set flag *flag* true if the specified *group* is alive less than the provided *percent*. If the *toggle* variable is present the flag will be set to false if the group is alive more than the specified *percent*. This process is stopped with *stopflag*.

groupName is the name of the group which the function checks if it is alive less than or not.

flag is a number corresponding to the flag that will set true if the group is alive less than the percent value

percent is the required percentage of a groups units that must be dead in order to set the flag true. Is evaluated x out of 100.

 stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

interval is an optional variable that allows you to specify how often (in seconds) the group_alive_less_than condition is checked. If not specified, interval defaults to 1.

toggle is an optional variable that if present will switch the *flag* value to false when the required conditions are not met. If not specified toggle defaults to false.

Examples:

```
mist.flagFunc.group_alive_less_than {
    units = 'myGroup,
    flag = 100,
    percent = 20,
    toggle = true,
}

Mission Editor Trigger: Switched Condition> Flag 100 is True> Message To
All('myGroup has less than 20 percent of its units remaining alive!')
```

mist.flagFunc.group_alive_more_than

mist.flagFunc.group_alive_more_than (table *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{

groupName = **string** *groupName*,

flag = **number/string** *flag*,

percent = **number** *percent*,

stopflag = **number/string** *stopflag*,

interval = **number** *interval*,

toggle = **boolean** *toggle*,

}

Once this function is run, it will start a continuously evaluated process that will set flag *flag* true if the specified *group* is alive more than the provided *percent*. If the *toggle* variable is present the flag will be set to false if the group is alive more than the specified *percent*. This process is stopped with *stopflag*.

groupName is the name of the group which the function checks if it is alive less than or not.

flag is a number corresponding to the flag that will set true if the group is alive less than the percent value

percent is the required percentage of a groups units that must be alive in order to set the flag true. Is evaluated x out of 100.

stopflag is an optional variable that allows you to specify a flag number that, if set true, will stop the process.

interval is an optional variable that allows you to specify how often (in seconds) the group_alive_more_than condition is checked. If not specified, interval defaults to 1.

toggle is an optional variable that if present will switch the *flag* value to false when the required conditions are not met. If not specified toggle defaults to false.

Examples:

```lua
mist.flagFunc.group_alive_more_than {
    units = 'myGroup,
    flag = 100,
    percent = 60,
    interval = 300,
    toggle = true,
}

Mission Editor Trigger: Switched Condition> Time Since Flag 100 is True, 5
seconds> Message To All('myGroup is still combat effective!')


--[[Once run, this function will start a process will repeat the message
"myGroup is still combat effective" every 5 minutes if more than 60 percent
of the group is alive.
]]
```

Scripting Functions

Mist's "scripting functions" are Lua functions that are designed for use by Lua programmers (as opposed to "flag functions" which were designed to be used with very little knowledge of Lua). Functions at the "mist" table level are Lua functions specifically designed for common DCS mission scripting tasks. "mist.utils" provides common Lua utilities. "mist.vec" provides Vec3 vector operations. "mist.debug" provides tools for viewing the global environment, outputting data to files, etc. "mist.demo" provides scripting demonstration functions.

General Scripting Functions

A set of DCS-specific scripting functions useful to mission Lua logic.

mist.scheduleFunction

number **mist.scheduleFunction (function** *f,* **table** *vars,* **number** *t, ,* **number** *rep,* **number** *st* **)**

An improvement over timer.scheduleFunction; this function can accept multiple variables, and optionally, a repetition rate and stop time.

This function schedules function *f* to run with the table *vars* unpacked (see the Lua unpack function in the Lua manual if you don't know what "unpacked" means) and passed to it at the specified time of *t*. It also returns a number (that can used for cancelling the scheduled function call with mist.removeFunction). The optional value *rep* is the time between repetitions of the function. *st* defines when the function will stop automatically. If no *vars* need to be passed to the function, just set *vars* equal to an empty table ({}).

Examples:

```
mist.scheduleFunction(trigger.action.setUserFlag, {101, true}, timer.getTime()
+ 40)
```

-Runs trigger.action.setUserFlag (101, true) one time, 40 seconds after the script is called.

```
local funcID = mist.scheduleFunction(main, {}, timer.getTime() + 10, 120)
```

-Runs the function "main" once every 10 seconds for the next two minutes. If you might want to intercede and stop this process, the integer ID of this process is returned into the variable "funcID". You could now stop the repetition of this function by using `mist.removeFunction(funcID)`.

mist.removeFunction

boolean mist.removeFunction (number *id*)

Removes the scheduled function with integer id *id* and returns true if a function was removed.

mist.addEventHandler

number mist.addEventHandler (function *handler*)

This is a simplified version of the simulator scripting engine's `world.addEventHandler` function. *handler* must a function that expects a single variable of a world simulator event. It also returns a number id for this event handler (for use with mist.removeEventHandler). For more information on world events, see the Simulator Scripting Engine documentation for world events.

Examples:

```
do
    activeWeapons = {}
```

```
    local function addWeapon(event)
        if event.id == world.event.S_EVENT_SHOT and world.event.weapon then
            activeWeapons[#activeWeapons + 1] = world.event.weapon
        end
    end

    mist.addEventHandler(addWeapon)
end
```

-Adds an event handler that, every time a weapon is fired, it adds that weapon to a global table that holds all weapon objects.

mist.removeEventHandler

boolean mist.removeEventHandler (number *id***)**

Removes event handler with id *id.*

mist.getUnitsInZones

table mist.getUnitsInZones (UnitNameTable *unit_names***, table** *zone_names***, string** *zone_type***)**

This function detects if any of the units listed inside the table of unit names *unit_names* are inside any of the zones listed in the table of zone names *zone_names*. The Unit objects of any in-zone units are returned in a (numerically indexed) table. The shape of the zone is determined by the optional variable *zone_type*. The following are allowed values for *zone_type*:

`'cylinder'` – cylindrical shaped zone extending to +/- infinity in altitude.
`'sphere'` – spherical zone.

See the code for mist.flagFunc.units_in_zones for a usage example.

mist.getUnitsInMovingZones

table mist.getUnitsInMovingZones (UnitNameTable *unit_names***, UnitNameTable** *zone_unit_names***, number** *radius***, string** *zone_type* **)**

This function detects if any of the units listed inside the table of unit names *unit_names* are inside any moving zones drawn around the locations of the units listed in the table of unit names *zone_unit_names*. The Unit objects of any in-zone units from *unit_names* are returned

in a (numerically indexed) table. The shape of the zone is determined by the optional variable *zone_type*. The following are allowed values for *zone_type*:

`'cylinder'` – cylindrical shaped zone extending to +/- infinity in altitude.
`'sphere'` – spherical zone.

See the code for mist.flagFunc.units_in_moving_zones for a usage example.

mist.pointInPolygon

boolean mist.pointInPolygon(Vec3 *point,* **table** *poly,* **number** *maxalt***)**

Returns true or false depending on if *point* is inside of a polygon defined by the points in the table *poly*. If *maxalt* is not specified the check assumes +/- infinity in altitude. Each point must be in either Vec2 or Vec3 format (see the simulator scripting engine pages on the ED wiki if you don't know what Vec2 or Vec3 means).The table poly needs to be in the following format:

[1] = {
      x = number,
      y = number,
      z = number or nil (nil if you're using Vec2),
},
[2] …

The polygon can be any convex shape. In the example image below note the points numbered from 1 to 5. The polygon is defined by "connecting the dots" sequentially. Any point within the red area is defined as inside of the polygon causing the function to return true.
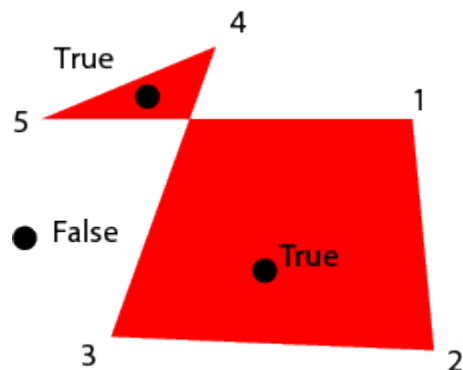
Also note that the order and position of points will change the shape of the object.
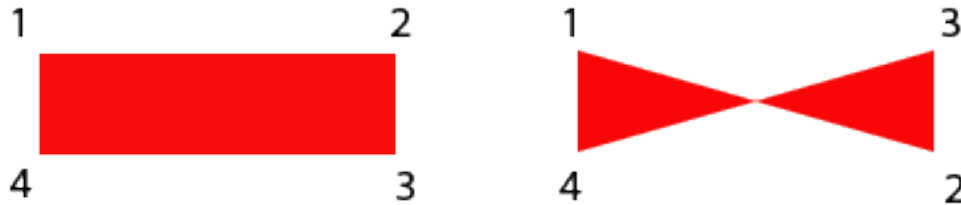
mist.getUnitsInPolygon

table **mist.getUnitsInPolygon (UnitNameTable** *unit_names***, table** *polyZone***, number** *maxAlt***)**

This function detects if any of the units listed inside the table of unit names *unit_names* are inside the polygon defined by *polyZone*. The Unit objects of any in-zone units are returned in a (numerically indexed) table. If *maxalt* is not specified the check assumes +/- infinity in altitude. Each point must be in either Vec2 or Vec3 format

mist.getDeadMapObjsInZones

table mist.getDeadMapObjsInZones(table *zone_names***)**

Returns a table of map objects indexed numerically that are dead within the zones defined in the table zone_names.

mist.getDeadMapObjsInPolygonZone

table mist.getDeadMapObjsInPolygonZone(table *zone***)**

Returns a table of map objects index numerically that are dead within the polygon zone defined by the points in the table *zone*.

mist.getUnitsLOS

table mist.getUnitsLOS(UnitNameTable *unitset1***, number** *altoffset1***, UnitNameTable** *unitset2***, number** *altoffset2***, number** *radius***)**

Returns a table with line of sight (LOS) data about which units in *unitset2* are LOS to the units in *unitset1*. The "sighting" point of each unit in each *unitset* is determined by the corresponding *altoffset* value. A maximum radius at which LOS is ignored can be optionally specified with *radius*.

The LOS data returned has the following format:

```
LOSData = {
```

```
    [1] = {
        unit = <a unitset1 "Unit"-type object>,
        vis = {
            [1] = <a unitset2 "Unit"-type object>,
            [2] = <a unitset2 "Unit"-type object>,
            ...
        }
    },
    [2] = {
        unit = <a unitset1 "Unit"-type object>,
        vis = {
            [1] = <a unitset2 "Unit"-type object>,
            [2] = <a unitset2 "Unit"-type object>,
            ...
        },
    },
    ...
}
```

Basically, each entry (which are indexed numerically) in the LOS data is a table that contains, at table key "unit", the unitset1 spotter Unit object, and at table key "vis", a table of all the unitset2 Unit objects that are visible to this unitset1 spotter unit.

A "real" returned LOS data might serialize to something like:

```
LOSData = {
    [1] = {
        unit = {id_ = 11553223},
        vis = {
            [1] = {id_ = 55523242},
        },
    },
    [2] = {
        unit = {id_ = 14522324},
        vis = {
            [1] = {id_ = 55523242},
            [2] = {id_ = 800242134},
            [3] = {id_ = 233452361},
        },
    },
}
```

mist.getNorthCorrection

number mist.getNorthCorrection (Vec3 *point*)

The map x direction differs from true north except on one line of longitude (33 degrees). This function returns the angle (in radians) between the map x axis and true north at *point*. This "correction" should

be added to any bearing computed from map Vec3/Vec2 coordinates to get the actual bearing (units are radians).

mist.getHeading

number mist.getHeading (Unit *unit*)

Returns the value of the specified *unit* heading (in radians).

mist.getPitch

number mist.getPitch (Unit *unit*)

Returns the value of the specified *unit* pitch (in radians).

mist.getRoll

number mist.getRoll (Unit *unit*)

Returns the value of the specified *unit* roll (in radians).

mist.getYaw

number mist.getYaw (Unit *unit*)

Returns the value of the specified *unit* yaw (in radians).

mist.getAoA

number mist.getAoA (Unit *unit*)

Returns the value of the specified *unit* Angle of Attack (in radians).

mist.getClimbAngle

number mist.getClimbAngle(Unit *unit*)

Returns the value of the specified *unit* Climb angle (in radians).

mist.getAttitude

table mist.getAttitude (Unit *unit*)

Returns a table of Heading, Pitch, Roll, Yaw, Angle of Attack, and Climb Angle of the specified *unit.* (see the code for mist.demos.printFlightData for a usage example).

mist.getAvgPos

table mist.getAvgPos(table *unitNames*)

Returns a vec3 coordinate of the averaged position defined between each unit passed by the table *unitNames*. With mist v4 this function now works with static objects

mist.getAvgPoint

table mist.getAvgPoint(table *points***)**

Returns a vec3 coordinate of the averaged position of the passed *points* table. Functionally is similar to mist.getAvgPos, but with vec2/3 points.

mist.tostringMGRS

string mist.tostringMGRS(table MGRS, number *n***)**

Returns a string of the MGRS coordinate specified in table *MGRS* to the accuracy of number *n.* Acceptable values for *n* are 0, 1, 2, 3, 4, or 5.

Example returned value:
mist.utils.tostringMGRS(table, 0) returns 38T AB
mist.utils.tostringMGRS(table, 3) returns 38T AB 123 123
mist.utils.tostringMGRS(table, 5) returns 38T AB 12345 12345

mist.tostringLL

string mist.tostringLL(number *lat***, number** *long,* **number** *n,* **string** *s***)**

Returns a string of the Latitude/Longitude coordinate specified in the numbers lat and long to the accuracy of number *n.* String s is an optional variable to display in the format of Degrees Minutes Seconds. If s is not present the default format will be degrees decimal minutes.

Example returned value:
mist.utils.tostringMGRS(table, 0) returns 38T AB
mist.utils.tostringMGRS(table, 3) returns 38T AB 123 123
mist.utils.tostringMGRS(table, 5) returns 38T AB 12345 12345

mist.tostringBR

string mist.tostringBR(number *az***, number** *dist,* **number** *alt,* **???** *metric***)**

Returns a string of the Bearing Range and Altitude (BRA) based on the inputed variables. The number *az* defines the bearing in radians. Range is defined by *dist*. Altitude is defined by the *alt*. If metric is not present the function will assume all values are in imperial units and will return the Range and Altitude in Nautical Miles and Feet. If metric is present the metric system will be used for these values.

mist.getMGRSString

string mist.getMGRSString(**table** *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **table** *UnitNameTable,*
acc = **number** *accuracy,*
}

Returns a string of the average position of units defined by a *UnitNameTable* in the MGRS format to the specified *accuracy.*

mist.getLLString

string mist.getLLString(**table** *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **table** *UnitNameTable,*
acc = **number** *accuracy,*
DMS = **???** *DMS,*
}

Returns a string of the average position of units defined by a *UnitNameTable* in the Latitude and Longitude format to the specified *accuracy*. If the optional variable *DMS* exists, the format will be in Degrees Minutes Seconds. If *DMS* is not present the format will be in Degrees Minutes Thousandths of Minutes.

mist.getBRString

string mist.getBRString(**table** *vars*)

vars has the following recognized fields (required entries in blue, optional in green):
{
units = **table** *UnitNameTable,*
ref = **table** *vec3,*
alt = **number** *altitude,*
metric = *metric,*
}

Returns a string in the Bearing Range Altitude (BRA) format of average position of units defined by a *UnitNameTable* from the reference point defined by a vec3 table *ref*. If metric is not present the function will assume all values are in imperial units and will return the Range and

Altitude in Nautical Miles and Feet. If metric is present the metric system will be used for these values.

mist.getLeadingPos

Vec3 **mist.getLeadingPos(table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

radius = **number** *radius,*

heading = **number** *heading OR* headingDegrees = **number** *headingDegrees*

}

Returns the Vec3 coordinates of the average position of the concentration of units most in the heading direction. The units are defined by the table *UnitNameTable* and the concentration is within the specified *radius.*

mist.getLeadingMGRSString

string **mist.getLeadingMGRSString(table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

radius = **number** *radius,*

heading = **number** *heading OR* headingDegrees = **number** *headingDegrees,*

acc = **number** *accuracy,*

}

Returns a string of the coordinates in MGRS format of the average position of the concentration of units most in the heading direction to the defined *accuracy*. The units are defined by the table *UnitNameTable* and the concentration is within the specified *radius.*

mist.getLeadingLLString

string **mist.getLeadingLLString(table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

radius = **number** *radius,*

heading = **number** *heading OR* headingDegrees = **number** *headingDegrees,*

DMS = **???** *DMS,*

}

Returns a string of the coordinates in Latitude and Longitude format of the average position of the concentration of units most in the heading direction to the defined *accuracy*. The units are defined by the table *UnitNameTable* and the concentration is within the specified *radius.* If the optional variable *DMS* exists, the format will be in Degrees Minutes Seconds. If *DMS* is not present the format will be in Degrees Minutes Thousandths of Minutes.

mist.getLeadingBRString

string **mist.getLeadingBRString(table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

radius = **number** *radius,*

heading = **number** *heading* OR headingDegrees = **number** *headingDegrees,*

ref = **table** *vec3,*

alt = **number** *altitude,*

metric = *metric,*

}

Returns a string in the Bearing Range Altitude (BRA) format of the concentration of units most in the heading direction. The units are defined by a *UnitNameTable.* The string is created based on the reference point defined by a vec3 table *ref*. If metric is not present the function will assume all values are in imperial units and will return the Range and Altitude in Nautical Miles and Feet. If metric is present the metric system will be used for these values.

mist.getMilString

string **mist.getMilString(number** *time***)**

Returns a string in military time format ranging from '0000' to '2359'. *time* is defined in seconds, if *time* is not passed the function will use the current time of the mission.

mist.getClockString

string **mist.getClockString(number** *time,* **boolean** *format***)**

Returns a string of the time in a clock format. *time* is defined in seconds, if *time* is not passed the function will use the current time of the mission. The boolean value *format* if present will return the string in a 12 Hour clock format with an AM or PM attached to the end of the string. By default the function returns the time in a 24 hour format. HH:MM:SS

mist.getDateString

string mist.getDateString(boolean *mString***, boolean** *order***, number** *time***)**

Returns a string of the date. The boolean *mString* defines if the month name is used in the string instead of the month digit. (false/nil for 3, true for 'March'). The boolean *order* defines if the date is presented in the date format mm/dd/yy or dd/mm/yy. *time* is defined in seconds, if *time* is not passed the function will use the current time of the mission, thus return the current date.

mist.isTerrainValid

boolean mist.isTerrainValid (table *vec2/vec3,* **table** *terrainTypes***)**

Returns true if the specified vec2 or vec3 coordinate is the correct type of terrain. The terrain types table accepts any of the following values:

```
LAND
SHALLOW_WATER
WATER
ROAD
RUNWAY
```

mist.terrainHeightDiff

number mist.terrainHeightDiff (table *vec2/vec3,* **number** *searchSize***)**

Returns the height difference in meters between the highest and lowest point in a given search area. The search area is centered at the *vec2/vec3* coordinate, and will search a max radius in meters defined by *searchSize*. If *searchSize* is not present search radius defaults to 5 meters. This functions is to be used to figured out if a point is to steep to spawn an object at.

mist.random

number mist.random (number/number *low/high,* **number** *high***)**

Returns a random number between the low and high values passed. If a single value is passed this value will act as the *high* number and a random number between 1 and high will be returned. If two values are passed the first variable is the *lowest* number that can be returned and the second variable is the *highest* number that can be returned.

mist.randomizeNumTable

table **mist.randomizeNumTable (table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

size = **number** *size,*

lowerLimit = **number** *lowerLimit,*

upperLimit = **number** *upperLimit,*

exclude = **table** *numbers,*

}

Returns a table of numbers (1 to size) in a randomized order of the specified *size*. The optional variables define which numbers will not be randomized within the table. The *lowerLimit* and *upperLimit* variables respectively define the low and upper limits on which numbers can be randomized. The *exclude* table is a table of numbers, in no particular order, that will not be randomized.

Example:

mist.randomizeNumTable({size = 10, lowerLimit = 3, upperLimit = 8}) could return

{1, 2, 6, 8, 5, 4, 3, 7, 9, 10}

mist.getRandomPointInCircle

table **mist.getRandPointInCircle (table** *vec2/vec3,* **number** *radius,* **number** *innerRadius***)**

Returns a randomly generated vec2 coordinate within the specified *radius* around the center *vec3 or vec 2* point given. If the optional variable *innerRadius* is given a random point will be generated that has a minimum distance of *innerRadius* and maximum distance of *radius* from the point.

mist.getRandomPointInZone

table **mist.getRandomPointInZone (string** *zoneName,* **number** *innerRadius***)**

Returns a randomly generated vec2 within a given trigger *zoneName*. . If the optional variable *innerRadius* is given a random point will be generated that has a minimum distance of *innerRadius* and maximum radius of the triggerzone as setup in the editor.

mist.getNextGroupId

number **mist.getNextGroupId ()**

Iterates and returns the next available groupId to be used with dynamically spawning AI. Note that this function is intended for your own use in spawning AI with coalition.addGroup. If you are going to use mist.dynAdd, this function is not needed. Additionally the value returned is accessible by the global value mist.nextGroupId.

mist.getNextUnitId

number **mist.getNextUnitId ()**

Iterates and returns the next available unitId to be used with dynamically spawning AI. Note that this function is intended for your own use in spawning AI with coalition.addGroup. If you are going to use mist.dynAdd, this function is not needed. Additionally the value returned is accessible by the global value mist.nextUnitId.

mist.stringMatch

boolean **mist.stringMatch (string** *string1,* **string** *string2,* **boolean** *caseSensitive***)**

Returns a boolean value of string1 and string2 hold the same data. Removes special lua characters on both strings to more easily check if the values are the same. If the boolean value *caseSensitive* is present the two strings much match the same case. If not present case sensitivity will not matter. Removes the following characters:

- ( ) _ [ ] . , # { } $ % ? + ^

It also removes <u>spaces</u> between characters.

mist.matchString

boolean **mist.matchString (string** *string1,* **string** *string2,* **boolean** *caseSensitive***)**

Same function as mist.stringMatch(). Just another name to access it by.

Group Orders

mist.groupToRandomPoint

table **mist.groupToRandomPoint (table** *vars***)**

<u>vars has the following recognized fields (required entries in blue, optional in green):</u>

{
group = **table** *groupTable,*
point = **table** *Vec3,*
radius = **number** *radius,*

form = **string** *formationName,*

heading = **number** *heading,*

headingDegrees = **number** *headingDegrees,*

speed = **number** *speed,*

disableRoads = **boolean** *disableRoads*

}

Creates a path for the specified groupTable from the groups current location to the Vec3 destination point.

Radius specifies the maximum distance from point for the last waypoint. If radius is greater than 0 a random point will be generated within the radius. If no radius is given it will default to 0.

form specifies the formation used while off road enroute to the waypoint. Formations default to the formation "cone" as defined in mist.ground.buildWP.

heading is the groups final orientation in radians. If no heading is given the final heading will be random.

headingDegrees is the groups final orientation in degrees. If no heading is given the final heading will be random.

Note: If both heading and headingDegrees are specified, headingDegrees will be used.

speed is the speed in meters per second the group is to travel at. If no speed is given the speed used at each waypoint will vary between 20 and 60 kilometers per hour.

disableRoads will allow or deny the group use of roads to get to the destination. If roads are not disabled and a radius

mist.groupRandomDistSelf

table **mist.groupRandomDistSelf (???** *group,* **number** *distance,* **string** *form,* **number** *heading,* **number** *speed***)**

Function will set the task of the specified group to go a random distance and direction from its current location. group can be either a group name or a group table. *Distance* is the max distance from the current location. If not specified a random distance between 300 and 1000 meters will be defined. *form* is the default formation name in as defined by the scripting engine. See mist.ground.buildWP or the Scripting Engine wiki for details. *heading* is the final heading the group will be oriented in once it reaches its destination. *speed* is the speed in kilometers per hour that the group will travel at to reach its destination.

mist.groupToRandomZone

table mist.groupToRandomZone (??? *group*, ??? *zone*, string *form*, number *heading*, number *speed*)

Function will set the task of the specified group to go a random zone as defined by zone. group can be either a group name or a group table. *zone* accepts a zone name, zone table, or table of zone names. If multiple zone names are given the function will randomly pick a zone to go to and pick at random a point in that zone as the final destination. *form* is the default formation name in as defined by the scripting engine. See mist.ground.buildWP or the Scripting Engine wiki for details. *heading* is the final heading the group will be oriented in once it reaches its destination. *speed* is the speed in kilometers per hour that the group will travel at to reach its destination.

Examples:
mist.groupToRandomZone(Group.getByName('group1'), trigger.misc.getZone('zone1'))
mist.groupToRandomZone(Group.getByName('group1'), 'zone1')
mist.groupToRandomZone('group1', {'zone1'})
--[[These three functions do the exact same thing]]

mist.groupToRandomZone('group1', {'zone1', 'zone2', 'zone3'})
--[[Group 1 will travel to a random point in one of the 3 zones]]

mist.groupToPoint

table mist.groupToPoint (??? *group*, ??? *point*, string *form*, number *heading*, number *speed*, boolean *useRoads*)

Function will set the task of the specified group to go a the point defined by a zones location. group can be either a group name or a group table. *point* can be a zoneTable or zoneName. If not specified a random distance between 300 and 1000 meters will be defined. *form* is the default formation name in as defined by the scripting engine. See mist.ground.buildWP or the Scripting Engine wiki for details. *heading* is the final heading the group will be oriented in once it reaches its destination. *speed* is the speed in kilometers per hour that the group will travel at to reach its destination. useRoads defines if the group will use a road to get to the point.

mist.goRoute

table mist.goRoute(table *group*, table *path*)

Function uses Controller.setTask using the table path to define the route and waypoint actions for the specified group.

Group Spawning and Data
Formerly the "SpawnCloneTeleport" Script

mist.dynAdd

table **mist.dynAdd (table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **table** *unitsTable,*
country = **string/number** *countryName/countryIndex,*
category = **string/number** *categoryName/categoryIndex,*
name **or** groupName = **string** *groupsName,*
groupId = **number** *groupId,*
clone = **anything** *clone,*
}

With valid data, this function will dynamically spawn a group consisting of *unitsTable* within the *categoryName* for *countryName* using the built in scripting function coalition.addGroup. **This function does not current support adding static objects, it however is a planned feature for the near future.** The function will also add the new groups data to mist.DBs as needed. Much of the data in this function matches the mission editor format of a groupTable. This function is "overloaded" with variables so that multiple other functions and formats and can generate some data as need to properly dynamically spawn a group. This function is used by all of the other group spawning, cloning and teleporting functions. Be mindful of the data you pass it! The scripting engine does not validate coordinates for you and you can spawn land objects on water or ships on land if you tell it to. mist.isTerrainValid was included to help in this regard.

See *unitsTable* description below as it has its own table requirements and optional variables.

If *groupId* is not specified the function will generate a new groupId based on the number of groups currently in the mission.

If a *groupsName* is not specified the function will generate a name based on the country the group belongs to, its category, and its group Id. Example: "USA gnd 25"

If *clone* is specified it will override any groupName, groupId, unitName, and unitId passes.

vars.units must be indexed numerically and matches the mission editor format of a groups contents.
vars.units = {

```
        [1] = {unitVars},
        [2] = {unitVars},
        ...
    },
```

unitVars has the following recognized fields (required entries in blue, optional in green):

```
{
x = number vec2XCoord,
y =  number vec2YCoord,
type = string objectTypeName,
unitId = number unitId,
name or unitName = string unitsName,
payload = table mePayloadTable,
speed = number speed,
alt = number altitude,
alt_type = string altitudeType,
skill = string skill,
route = table routeData,
callsign = table callsignTable,
livery_id = string liveryName,
}
```

A unit of *objectTypeName* will spawn at the *vec2XCoord* and *vec2YCoord* coordinates.

If *unitId* is not specified the function will generate a new unitId based on the number of units current in the mission.

If a *unitsName* is not specified the function will generate a new unit name based on the group's name indexed numerically. Example: "USA gnd 25 unit 1"

speed, *altitude*, *altitudeType*, and *payload* only matter for planes and helicopter aircraft types.

speed is measured in Meters Per Second. If *speed* is not specified and the group is a helicopter or plane, it will default to 60 mps for helicopters and 150 mps for airplanes.

altitudeType is a string of the type of altitude measurement used. Valid entries are 'RADIO' for Above Ground Level and 'BARO' for Above Sea level. If not specified, the function will default to 'RADIO' to avoid spawning underground.

altitude is measured in meters and is dependent on the *altitudeType* specified. If no altitude is specified and the group is a helicopter or an airplane the altitude will default to 500 meters for helicopters and 2000 meters for airplanes.

payload is a table matching the mission editor table for a payload. It is important to note that the scripting engine cannot generate the values needed for this table unless the information is provided beforehand. If the unit being spawned is an helicopter or airplane and no payload is specified, then the aircraft WILL NOT HAVE FUEL and its engines will not run.

route is an optional table that if present, will embed the route data into the group as it spawns so that it will start its route data immediately as it spawns. The route table should be in the following format.
route.points = {
       [1] = {pointTable},
       [2] = {pointTable},
       ...
},

The function is overloaded to also accept a table in this format:
route = {
       [1] = {pointTable},
       [2] = {pointTable},
       ...
},
liveryName is a optional string variable that defines the texture livery the unit will spawn with. If not specified it defaults to the default texture of the country for the aircraft type.

callsign is an optional table variable that will define the callsign used by an aircraft or helicopter group. If not specified the scripting engine (not mist) picks one seemingly at random.

skill is an optional string variable that defines what skill level the unit will be. If not specified the skill level is set to '*random*'.

mist.dynAddStatic

table mist.dynAddStatic (table *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{

country = **string/number** *countryName/countryIndex,*

category = **string/number** *categoryName/categoryIndex,*

x = **number** *vec2XCoord,*

y = **number** *vec2YCoord,*

type = **string** *objectTypeName,*

name or groupName = **string** *groupsName,*

groupId = **number** *groupId,*

clone = **anything** *clone,*

dead = ***boolean*** *dead,*

heading = ***number*** *heading,*

}

With valid data, this function will dynamically spawn a static object of the specified *type* within the *categoryName* for *countryName* using the built in scripting function coalition.addStaticGroup. The function will also add the new static objects data to mist.DBs as needed. This function is "overloaded" with variables so that multiple other functions and formats and can generate some data as need to properly dynamically spawn the static object. This function is used by all of the other group spawning, cloning and teleporting functions if those functions are passed a static object. Be mindful of the data you pass it! The scripting engine does not validate coordinates for you and you can spawn land objects on water or ships on land if you tell it to. mist.isTerrainValid was included to help in this regard.

A static object of *objectTypeName* will spawn at the *vec2XCoord* and *vec2YCoord* coordinates.

If *groupId* is not specified the function will generate a new groupId based on the number of groups currently in the mission.

If a *groupsName* is not specified the function will generate a name based on the country the group belongs to, its category, and its group Id. Example: "USA gnd 25"

If *clone* is specified it will override any groupName, groupId, unitName, and unitId passes.

If *dead* is specified the object will be rendered as already destroyed when spawned in.

If *heading* is passed this object be oriented in the specified heading in degrees. If not present the *heading* will be random between 0 and 359.

mist.groupTableCheck

boolean **mist.groupTableCheck (table** *groupTable***)**

Checks the passed *groupTable* to see if required variables are missing to spawn the group via mist.dynAdd() If no variables are missing the function will return true.

mist.respawnGroup

table **mist.respawnGroup (string** *groupName,* **boolean/number** *task***)**

Respawns a group of the name *groupName* at its initial location as set in the mission editor. If the optional *task* variable is provided the group will automatically be given its default task. If *task* is a number the new task will be spawned the value of the number in seconds after the group spawns. If it is any other data type, the task will be embedded into the group to spawn. If *task* is not provided group will remain stationary or RTB if an aircraft. Function returns a mission editor formatted table of the group.

mist.cloneGroup

table **mist.cloneGroup (string** *groupName,* **boolean/number** *task***)**

Clones a group of the name *groupName* at its initial location as set in the mission editor. If the optional *task* variable is provided the group will automatically be given its default task. If *task* is a number the new task will be spawned the value of the number in seconds after the group spawns. If it is any other data type, the task will be embedded into the group to spawn. If *task* is not provided group will remain stationary or RTB if an aircraft. Function returns a mission editor formatted table of the group.

mist.teleportGroup

table **mist.teleportGroup (string** *groupName,* **boolean/number** *task***)**

Teleports a group of the name *groupName* at its initial location as set in the mission editor. If the optional *task* variable is provided the group will automatically be given its default task. If *task* is a number the new task will be spawned the value of the number in seconds after the group spawns. If it is any other data type, the task will be embedded into the group to spawn. If *task* is not provided group will remain stationary or RTB if an aircraft. Function returns a mission editor formatted table of the group.

mist.respawnInZone

table mist.respawnInZone (**string** *groupName,* **string** *zoneName,* **Boolean** *disperse,* **number** *radius*)

Respawns a pre-existing group of the *groupName* at a random location inside of *zoneName*. Function will completely respawn the given group. *zoneName* also accepts a table of zone names which it will pick from at random to spawn the group. If the optional variable *disperse* is provided each unit in the group will spawn at a random location within a given *radius*. If *radius* is not provided a max distance of 250 meters will be used.

mist.teleportInZone

table mist.teleportInZone (**string** *groupName,* **string** *zoneName,* **Boolean** *disperse,* **number** *radius*)

Teleports a currently spawned group of the *groupName* at a random location inside of *zoneName*. Only alive units will be teleported. *zoneName* also accepts a table of zone names which it will pick from at random to spawn the group. If the optional variable *disperse* is provided each unit in the group will spawn at a random location within a given *radius*. If *radius* is not provided a max distance of 250 meters will be used.

mist.cloneInZone

table mist.cloneInZone (**string** *groupName,* **string** *zoneName,* **Boolean** *disperse,* **number** *distance*)

Clones a pre-existing group of the *groupName* at a random location inside of *zoneName*. Function will clone a specified group. *zoneName* also accepts a table of zone names which it will pick from at random to spawn the group. If the optional variable *disperse* is provided each unit in the group will spawn at a random location within a given *radius*. If *radius* is not provided a max distance of 250 meters will be used.

mist.teleportToPoint

table mist.teleportToPoint (**table** *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{
point = **vec3** *point,*
gpName = **string** *groupName,*
action = **string** *action,*
disperse = **boolean** *anything,*

maxDisp = **number** *distance*,

radius = **number** *distance*,

innerRadius = **number** *distance*,

}

Performs the specified action on an existing group of gpName to the specified point. If the optional disperse variable is given each unit in the group will be spawned within the distance specified by maxDisp. If disperse is not provided the group will spawn in the formation as defined by the mission editor. When a radius is given the group will spawn at a random point within the given distance. The variable innerRadius specifies a minimum distance from the point.

Valid actions are the following:

String teleport

Teleports the group in its current state (dead units will not respawn)

String respawn

Respawns the group

String clone

Clones the group

Example:

```
local vars = {}
vars.gpName = 'group1'
vars.action = 'clone'
vars.point = vec3
vars.radius = 1000
vars.disperse = 'disp'
vars.maxDisp = 500
mist.teleportToPoint(vars)
```


mist.spawnRandomizedGroup

boolean **mist.spawnRandomizedGroup (string** *groupName,* **table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

lowerLimit = **number** *lowerLimit,*

upperLimit = **number** *upperLimit,*

excludeNum = **table** *numbers,*

excludeType= **table** *unitTypes,*

}

Respawns a group of the name *groupName* at its initial location as set in the mission editor but randomizes the order of the units within the group. Group will spawn with the same task as before. The optional variables define which units in the group will not be randomized. The *lowerLimit* and *upperLimit* variables respectively define the low and upper limits on which unit indexes can be randomized. The *excludeNum* table is a table of numbers, in no particular order, of which unit indexes will be ignored. The *excludeType* table defines the unitType name of units that will be ignored.

Example:

mist.spawnRandomizedGroup('tanks', {excludeType = {'T-80U', 'GAZ-66'}})

Each T-80U and GAZ-66 will be in the same position within the group as defined in the mission editor. Every single other unit in the group could be in a randomized order.

Keep in mind unit properties such as unitId and unitName stay with the unit.

mist.randomizeGroupOrder

table **mist.randomizeGroupOrder (table** *units,* **table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):
{
lowerLimit = **number** *lowerLimit,*
upperLimit = **number** *upperLimit,*
excludeNum = **table** *numbers,*
excludeType= **table** *unitTypes,*
}
Returns the table of passed *units* in a randomized order within the group as defined by the optional *vars* table. The optional variables define which units in the group will not be randomized. The *lowerLimit* and *upperLimit* variables respectively define the low and upper limits on which unit indexes can be randomized. The *excludeNum* table is a table of numbers, in no particular order, of which unit indexes will be ignored. The *excludeType* table defines the unitType name of units that will be ignored.

Example:

mist.spawnRandomizedGroup('tanks', {excludeType = {'T-80U', 'GAZ-66'}})

Each T-80U and GAZ-66 will be in the same position within the group as defined in the mission editor. Every single other unit in the group could be in a randomized order.

Keep in mind unit properties such as unitId and unitName stay with the unit.

mist.getGroupData

table mist.getGroupData (string *groupName***)**

Returns a table in the format required to spawn the group. This function searches mist.DBs.groupsByName for the specified groupName.

mist.getCurrentGroupData

table mist.getCurrentData (string *groupName***)**

Returns a table in the format required to spawn the group with the specified *groupName*. Dead units will be ignored. Skill level will be randomized.

mist.getPayload

table mist.getPayload (string/mumber *unitName/unitId***)**

Returns a table of the payload set for *unitName* in the mission editor. "Live" payload is not currently possible.

mist.getGroupPayload

table mist.getPayload (string/number *groupName/groupId***)**

Returns a table of the payload for each unit in the specified *groupName* as defined in the mission editor. "Live" payload is not currently possible.

mist.getUnitSkill

string mist.getUnitSkill (string *unitName***)**

Returns a string of skill for the specified *unitName* as defined in the mission editor or dynamically spawned groups. Returns false if no skill is found.

mist.getGroupPoints

table mist.getGroupPoints (string/number *groupName/groupId***)**

Returns a table of Vec2 points that define the default route of the group named *groupName* as set within the mission editor. This function is great when used in combination with mist.pointInPolygon;

you can "draw" your polygon with the waypoints of a unit (that never gets activated), and then feed the mist.pointInPolygon with the returned values of mist.getGroupPoints.

Example returned value:

 { [1] = {x = 299435.224, y = -1146632.6773}, [2] = { x = 663324.6563, y = 322424.1112}}

Example (used in conjunction with mist.pointInPolygon):

```
inZone = mist.pointInPolygon(point, mist.getGroupPoints('Polygon Group 1'))
```

- assuming that "point" is a vec2 or vec3 map coordinate, inZone will now be a boolean that reflects whether or not that point is inside the polygon created by the waypoints of the group named "Polygon Group 1"

mist.getGroupRoute

table mist.getGroupRoute (string/number *groupName/groupId,* **boolean** *task***)**

Returns a table of the necessary data that defines the default route of group named *groupName* as set in the mission editor. Similar to mist.getGroupPoints but returns a table with additional values for group formation, speed, altitude, and altitude type. If *task* is true the function will get all task actions assigned to the route.

mist.getLeadPos

table mist.getLeadPos(table/string *group***)**

Returns a vec3 coordinate of the first unit in a specified *group*. *group* can be either a group table or the group name.

Message

Mission Scripting messaging system. Capable of displaying multiple messages via outText on screen at a single time. Messages get sent to the specified groups and display for the specified time. The MiST code locks the message display box refresh rate to 0.1 seconds.

Important Note: Due to how the message system operates it requires players to be an active member of a coalition in order to receive and view messages. Additionally due to how the simulation handles Combined Arms the message system sends each message twice. First to coalition so Combined Arms players can see the message, and again to individual player aircraft. Unlike individual aircraft, Combined Arms players cannot see personalized messages.

mist.message.add

number mist.message.add (table *vars***)**

{

text = **string** *text,*
displayTime = **number** *displayTime,*
msgFor = **table** *msgFor,*
name = **string** *name,*
sound = **string** *filename,*
multSound = **table** *multSound,*

}

Adds *text* to the message queue for the duration of *displayTime* to the applicable groups designated by *msgFor*. If the optional variable sound is used and the *filename* is a valid sound file embedded into the mission, the message will play the sound file once for the receiving groups. Using the optional variable *multSound* allows for sounds to be played at different times during the message. The optional variable *name* is used to name the message, if another message already exists with the same name it will automatically replace the older message. This is useful in updating an existing message to output new data, for example coordinates of a moving target. Function returns a number value of the message ID to be used with mist.message.remove. Messages will automatically be removed once the message display time has been reached.

table *msgFor,* accepts a string or a table of strings as keywords to define which groups will receive the message. This table can be use a variety of values to send the message to a very specific group of players. Acceptable values are in table format similar to the following:

    {
            units = {...}, -- unit names.
            coa = {...}, -- coa names
            countries = {...}, -- country names
            CA = {...}, -- looks just like coa.
            unitTypes = { red = {TYPENAME}, blue = {}, all = {}, Russia = {},}
    }

Examples:

- msg.msgFor = {coa = {'all'}} will send the message to all players including CA players
- msg.msgFor = {countries = {'Russia', "Georgia'}} will send the message to all players in Russian or Georgian aircraft. (no CA player receives a message)
- msg.msgFor = {countries = {'Russia', "Georgia'}, CA = 'red'} will send the message to all players in Russian or Georgian aircraft and red CA roles.
- msg.msgFor = {unitTypes = {red = {'Su-25T'}}} will send to the message to all players on the red team in Su-25Ts
- msg.msgFor = {unitTypes = {all = {'Su-25T'}}} will send to the message to all players in Su-25Ts
- msg.msgFor = {unitTypes = {Russia = {'Su-25T'}, Georgia = {'Su-25T'}}} will send to the message to all players in Su-25Ts that belong to Georgia and Russia

- msg.msgFor = {unitTypes = {Russia = {'Su-25T'}, Georgia = {'Su-25T'}}, CA = {'red'}} will send to the message to all players in Su-25Ts that belong to Georgia and Russia and to Combined Arms players on red.

Script Example:

local msg = {}

msg.text = 'Hello World'

msg.displayTime = 25

msg.msgFor = {coa = {'all'}}

mist.message.add(msg)

The multSound table contains a number of tables embedded within to define the sound files to be played and the time after the message started to play each sound file. Each sound file is played once. Note that DCS automatically stops any currently playing audio file when a new file is triggered so that files do not "step over" one another.

```
{
        [1] = {
                time = number timeToPlayAudioFile
                file = string fileName
        },
        [2] ... etc
}
```

Example:

local msg = {}

msg.text = 'Tusk, this is Wizard. New Picture, area threats include enemy fighter aircraft, over.'

msg.displayTime = 50

msg.msgFor = { coa = {'all'}}

msg.multSound = {

[1] = {time = 0, file = 'hq_Tusk.wav'},

[2] = {time = 1.5, file = 'hq_ThisIsWizard.wav'},

[3] = {time = 4, file = 'hq_newPicture.wav'},

[4] = {time = 4.5, file = 'hq_AreaThreatsInclude.wav'},

[5] = {time = 6, file = 'hq_EnemyFighterAircraft.wav'},

[6] = {time = 8, file = 'hq_over.wav'},

}

mist.message.add(msg)

mist.message.removeById

boolean **mist.message.removeById (number** *id***)**

Removes the message with integer *id* from the list of messages. If the message was removed the function will return true. NOTE: renamed from mist.message.remove in Mist v3.0! Change function names as needed!

mist.message.remove

boolean mist.message.remove (table *self***)**

Removes the message if passed the message table. If the message was removed the function will return true. Function is for internal mist usage.

General Purpose Message Functions

The following functions will output assorted text via mist.message.add along with performing other mist functions or useful functions.

mist.msgMGRS

string mist.msgMGRS (table *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **table** *UnitNameTable,*
acc = **number** *accuracy,*
displayTime = **number** *displayTime,*
msgFor = **table** *recipients,*
text = **string** *text,*
}

Utilizes mist.getMGRSString and mist.message.add functions to display a coordinates at the specified *accuracy* via the mist message system in the MGRS format to the specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message.

mist.msgLL

string mist.msgLL (table *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **table** *UnitNameTable,*
acc = **number** *accuracy,*

DMS = **boolean** *DMS,*

text = **string** *text,*

displayTime = **number** *displayTime,*

msgFor = **table** *recipients,*

}

Utilizes mist.getLLString and mist.message.add functions to display a coordinates at the specified *accuracy* via the mist message system in the Longitude/Latitude format to the specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message. If true *DMS* will set the coordinates in Degrees Minutes Seconds. If False format will be Degrees Minutes Thousandths.

mist.msgBR

string **mist.msgBR (table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

ref = **table** *vec2/vec3,*

alt = **boolean** *altitude,*

metric = **boolean** *metric,*

text = **string** *text,*

displayTime = **number** *displayTime,*

msgFor = **table** *recipients,*

}

Utilizes mist.getBRString and mist.message.add functions to display a coordinates at the specified *accuracy* via the mist message system in the Bearing Range format from the *reference vec2 or vec3* point to the *units*. The message will display to the specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message. If *altitude* is true message will also contain the altitude in "Bearing Range Altitude" format. If *metric* is true, the message will use the metric system.

mist.msgBRA

string **mist.msgBRA (table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **table** *UnitNameTable,*

ref = **string** *unitName,*

metric = **boolean** *metric,*

text = **string** *text,*

displayTime = **number** *displayTime,*

msgFor = **table** *recipients,*
}

Utilizes mist.getBRString and mist.message.add functions to display a coordinates at the specified *accuracy* via the mist message system in the Bearing Range Altitude format from the *unitName* to the average position of *units*. The message will be displayed to the specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message. If *metric* is true, the message will use the metric system.

mist.msgBullseye

string mist.msgBullseye (table *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{
units = **table** *UnitNameTable,*

ref = **string** *coalition,*

metric = **boolean** *metric,*

displayTime = **number** *displayTime,*

msgFor = **table** *recipients,*

text = **string** *text,*
}

Utilizes mist.getBRString and mist.message.add functions to display a coordinates at the specified *accuracy* via the mist message system in the Bearing Range Altitude format from the *coalition* Bullseye point to the average position of *units*. The message will be displayed to the specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message. If *metric* is true, the message will use the metric system.

Ref variable accepts 'red' or 'blue'. Case does not matter.

mist.msgLeadingMGRS

string mist.msgLeadingMGRS(**table** *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

radius = **number** *radius,*

heading = **number** *heading OR* headingDegrees = **number** *headingDegrees,*

acc = **number** *accuracy,*

displayTime = **number** *displayTime,*

msgFor = **table** *recipients,*

text = **string** *text,*

}

Utilizes mist.getLeadingMGRS and mist.message.add functions to display a coordinate in MGRS format of the average position of the concentration of units most in the heading direction to the defined *accuracy*. The units are defined by the table *UnitNameTable* and the concentration is within the specified *radius.* The message will be displayed to the specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message.

mist.msgLeadingLL

string mist.msgLeadingLL(**table** *vars*)

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

radius = **number** *radius,*

heading = **number** *heading OR* headingDegrees = **number** *headingDegrees,*

DMS = **???** *DMS,*

displayTime = **number** *displayTime,*

msgFor = **table** *recipients,*

text = **string** *text,*

}

Utilizes mist.getLeadingLLString and mist.message.add functions to display the coordinates in the Latitude and Longitude format of the average position of the concentration of units most in the heading direction to the defined *accuracy*. The units are defined by the table *UnitNameTable* and the concentration is within the specified *radius.* If the optional variable *DMS* exists, the format will be in Degrees Minutes Seconds. If *DMS* is not present the format will be in Degrees Minutes Thousandths of Minutes. The message will be displayed to the

specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message.

mist.msgLeadingBR

string **mist.msgLeadingBR(table** *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

{

units = **table** *UnitNameTable,*

radius = **number** *radius,*

heading = **number** *heading* **OR** headingDegrees = **number** *headingDegrees,*

ref = **table** *vec3,*

alt = **number** *altitude,*

metric =**boolean** *metric,*

}

Utilizes mist.getLeadingLLString and mist.message.add functions to display the coordinates in the Bearing Range Altitude (BRA) format of the concentration of units most in the heading direction. The units are defined by a *UnitNameTable.* The string is created based on the reference point defined by a vec3 table *ref*. If metric is not present the function will assume all values are in imperial units and will return the Range and Altitude in Nautical Miles and Feet. If metric is present the metric system will be used for these values. The message will be displayed to the specified *recipients* for the given *displayTime*. If *text* is provided, the coordinates will be added to the end of the text message.

fixedWing

mist.fixedWing.buildWP

table **mist.fixedWing.buildWP (table** *point,* **string** *type,* **number** *speed,* **number** *altitude,* **string** *altitudeType***)**

Returns a table of a valid waypoint entry that is defined by the vec2 or vec3 coordinate of *point*. *Type* defines the turn method used on the waypoint. If no turn method is specified the default "turning point" will be used. The optional variable *speed* is a numerical value in meters per second for the current waypoint. If no *speed* is specified the mission editor default of 500 kilometers per hour will be used. The optional *altitude* variable will set the altitude which a waypoint will occur at. If no *altitude* is specified the mission editor default of 2000 meters will be chosen. String *altitudeType* defines whether the waypoint will measured Above Ground Level or Above Sea level. If no *altitudeType* is specified the

script will default to Above Ground Level. NOTE: This differs from the DCS Mission Editor which defaults to Above Sea Level.

Acceptable values for type and altitudetype are described in the Simulator Scripting Engine wiki page.

Type:
flyOverPoint - Aircraft will fly over the WP before moving on to next WP
turningpoint - Aircraft perform a lead turn ahead of the WP before moving on to next WP
Altitude Type:
Baro (ASL) - The altitude will be measured Above Sea Level
Radio (AGL) - The altitude will be measured Above Ground Level

Heli

mist.heli.buildWP

table mist.heli.buildWP (table *point,* **string** *type,* **number** *speed,* **number** *altitude,* **string** *altitudeType***)**

Returns a table of a valid waypoint entry that is defined by the vec2 or vec3 coordinate of *point*. *Type* defines the turn method used on the waypoint. If no turn method is specified the default "fin point" will be used. The optional variable *speed* is a numerical value in meters per second for the current waypoint. If no *speed* is specified the mission editor default of *200 kilometers per hour* will be used. The optional *altitude* variable will set the altitude which a waypoint will occur at. If no *altitude* is specified the mission editor default of *500* meters will be chosen. String *altitudeType* defines whether the waypoint will measured Above Ground Level or Above Sea level. If no *altitudeType* is specified the script will default to Above Ground Level. NOTE: This differs from the DCS Mission Editor which defaults to Above Sea Level.

Ground

Mission Scripting messaging system. Capable of displaying multiple messages via outText on screen at a single time. Messages get sent to the specified groups and display for the specified time.

mist.ground.buildWP

table mist.ground.buildWP (table *point,* **string** *formation,* **number** *speed***)**

Returns a table of a valid waypoint entry that is defined by the vec2 or vec3 coordinate of *point*. The optional varibale string *formation* defines the type of formation used with the waypoint, if no formation is given the *formation* will default to "cone". The optional variable *speed* takes a numerical

value in meters per second and assigns it to the waypoint. If speed is not defined the mission editor default of 20 kilometers per hour will be used.

Acceptable formations as described in the Simulator Scripting Engine Wiki page.

> "Off Road" - moving off-road in Column formation
> "On Road" - moving on road in Column formation
> "Rank" - moving off road in Row formation
> "Cone" - moving in Wedge formation
> "Vee" - moving in Vee formation
> "Diamond" - moving in Diamond formation
> "EchelonL" - moving in Echelon Left formation
> "EchelonR" - moving in Echelon Right formation

Code Example:

```
local path = {}
path[#path + 1] = mist.ground.buildWP(startPoint, 'Diamond', 5)
path[#path + 1] = mist.ground.buildWP(endPoint, 'Diamond', 5)
```

mist.ground.patrol

nothing mist.ground.patrol (table/string *groupTable/groupName, ,* **string** *patrolType,* **string** *formation,* **number** *speed***)**

This function will re-assign the route of the specified *groupName* as defined in the mission editor when the group reaches the end of their route effectively creating a patrol. If the optional variable *patrolType* is specified as 'doubleBack' the group will double back from the last waypoint to the first before starting the patrol again. Optional variable *formation* is a string of valid formations that the group will use for all waypoints that are not "on road". The optional variable of *speed* is the speed the group will travel at in meters per second.

Code Example:

```
mist.ground.patrol('myGroup', nil, 'diamond', 10)
```

mist.ground.patrolRoute

nothing mist.ground.patrolRoute (table *vars***)**

vars has the following recognized fields (required entries in blue, optional in green):

```
{
gpData = string/table groupName/groupTable,
useGroupRoute = string groupName,
speed = number speed,
```

offRoadForm = **string** *offRoadForm,*

onRoadForm = **string** *onRoadForm,*

pType = **string** *patrolType,*

route = **table** *routeTable,*

}

This function will re-assign a route for *groupName* once the group reaches the end if its route. If the optonal variable *useGroupRoute* is specified, the groups route as defined in the mission editor will be assigned to group corresponding with groupName. If the optional variable *pType* is specified as 'doubleBack' the group will double back from the last waypoint to the first before starting the patrol again. Optional variable *offRoadForm* is a string of valid formations that the group will use for all waypoints that are not "on road". Optional variable onRoadForm is a string of valid formations that the group will use for all on road waypoints. The optional variable of *speed* is the speed the group will travel at in meters per second.

If the optional entry route is specified, this function will use the passed route and will ignore all other optional variables.

Important Note concerning mist.ground.patrol and mist.ground.patrolRoute

These functions are susceptible to strange AI movement bugs. Specifically if you place the first and second waypoint to close to each other or simply have a single "on road" waypoint the AI might stop moving . Also certain types of turns with certain types of formations can cause two AI units to get stuck in avoidance logic and appear to "drag race" each other.

Utils

Contains general Scripting utilities. May be DCS specific, or code that could be ported into any application that uses Lua.

mist.utils.makeVec2

table   **mist.utils.makeVec2 (table** *Vec3***)**

This function takes a *Vec3* table and returns it as a Vec2 table.

mist.utils.makeVec3

table   **mist.utils.makeVec3 (table** *vec2***,   number** *y* **)**

This function takes a *vec2* table and converts it into the Vec3 format. The *y* variable is optional and it specifies the altitude to be used in the vec3 table. If *y* is not set the value defaults to 0. The *vec2* table can also be in a WP format with an 'alt' variable being present.

mist.utils.makeVec3GL

table   **mist.utils.makeVec3GL (table** *vec2/vec3***, number** *offset* **)**

This function takes a *vec2* or a *vec3* table and converts it into the Vec3 format with vec3.y (altitude) at the ground level of the point. The *offset* variable is optional and it specifies the altitude above ground level to offset from the point. If *offset* is not set the value defaults to 0.

mist.utils.zoneToVec3

table   **mist.utils.makeVec3 (table** *zone* or **string** *zonename* **)**

This function takes a *zone* table or a string *zonename* and converts it into the Vec3 format. This is useful in using the position of a zone to define waypoints locations.

mist.utils.vecToWP

table   **mist.utils.vecToWP (table** *vec***)**

This function takes a *vec* table and converts it to a WP format. WP format is {x, y, alt}. The vec can be either vec2 or vec3. If vec2 is passed the alt value will be returned at ground level.

mist.utils.unitToWP

table   **mist.utils.unitToWP (Unit** *unit* or **string** *unitName***)**

This function takes a *unit* object or *unitName* and converts it's current position to a WP format. WP format is {x, y, alt, speed, alt_type}.

mist.utils.toDegree

number   **mist.utils.toDegree(number** *angle***)**

This function takes an *angle* in radians and converts it to degrees.

mist.utils.toRadian

number   **mist.utils.toRadian(number** *angle***)**

This function takes an *angle* in degrees and converts it to radians.

mist.utils.deepCopy

table   **mist.utils.deepCopy(table** *table***)**

This code is from http://lua-users.org/wiki/CopyTable.

This function returns a "deep copy" of *table*, "deep" in that the copy recursively progresses down all "levels" of the table, and the copy shares the same metatables as the passed-in table. This function also correctly handles tables with cycles. This function is quite often useful because tables are passed by reference in Lua, not by value.

English: If a variable is declared to equal a table, Lua does not create a new table, it instead creates a new reference for the table. So now you have two references to the **same** table and of course, any change made to one reference will apply to the other reference. **mist.utils.deepCopy** gets around this by creating an entirely new table that is effectively a "clone" of the other table.

mist.utils.round

number **mist.utils.round(number** *number,* **number** *idp***)**

This function takes a *number* and returns a rounded version of it. Optional *idp* defines how many places after the decimal to round the number to; for example, an *idp* of 2 means the number will be rounded to the nearest hundredth. *idp* can be negative too, for example, -3 would round to the nearest thousand. If not specified, *idp* defaults to 0 (rounds to the nearest whole number).

mist.utils.roundTbl

number **mist.utils.roundTbl(table** *table,* **number** *idp***)**

This function takes a *table* and returns a rounded version of the numbers that may be present within it. Note this function only rounds the values directly passed, if there are numbers within nested tables these values will not be rounded. Optional *idp* defines how many places after the decimal to round the number to; for example, an *idp* of 2 means the number will be rounded to the nearest hundredth. *idp* can be negative too, for example, -3 would round to the nearest thousand. If not specified, *idp* defaults to 0 (rounds to the nearest whole number).

mist.utils.dostring

boolean, ??? **mist.utils.dostring(string** *code***)**

Executes the string *code* as Lua code. The first variable returned is a boolean value indicating whether the code successfully compiled. If false, then this value will be the compilation error. If true, then any further variables returned will be whatever the code you executed returned.

mist.utils.basicSerialize

string mist.utils.basicSerialize(**value** *val*)

Returns tostring(*val*), unless *val* is nil or a string. In the case of nil, it returns an empty string (this will PROBABLY be changed to return 'nil' in future versions). In the case *val* is a string, it returns `string.format('%q', s)`.

mist.utils.serialize

string mist.utils.serialize(**string** *name*, **value** *t*)

Returns the string `name = <serialized value>` where `<serialized value>` is the value *t* serialized to a string. Typically, *t* will be a table. The resulting string has a very pleasing, readable look if displayed or output to a file. However, the function goes into an infinite loop if *t* contains cycles (cycles are tables that contain, somewhere within them, references to themselves), so be warned!

mist.utils.serializeWithCycles

string mist.utils.serializeWithCycles(**string** *name*, **value** *t*)

Returns the string `name = <serialized value>` where `<serialized value>` is the value *t* serialized to a string. Typically, *t* will be a table. Unlike mist.utils.serialize, *t* can contain cycles, however, the resulting serialized table is a little less readable (to human eyes).

mist.utils.oneLineSerialize

string mist.utils.oneLineSerialize(**table** *t*)

Returns a serialization of table *t* on a single line of text; *t* cannot contain cycles. If the table *t* is small, then the resulting string will be more readable than any other serialization method. It's great for outputting really small tables to dcs.log with print just to see what's in them.

mist.utils.tableShow

string mist.utils.tableShow(**table** *t*)

Returns a string that shows the contents of table *t*. THIS IS NOT A SERIALIZATION FUNCTION; unlike all the serialization functions, you CANNOT run the returned string with loadstring or dostring. This function is solely made for exploring the contents of a table.

mist.utils.metersToNM

number mist.utils.metersToNM(**number** *n*)

Returns the conversion of meters defined as number *n* to Nautical Miles.

mist.utils.metersToFeet

number **mist.utils.metersToFeet(number** *n***)**

Returns the conversion of meters defined as number *n* to Feet.

mist.utils.NMToMeters

number **mist.utils.NMToMeters(number** *n***)**

Returns the conversion of nautical miles defined as number *n* to meters.

mist.utils.feetToMeters

number **mist.utils.feetToMeters(number** *n***)**

Returns the conversion of feet defined as number *n* to meters

mist.utils.mpsToKnots

number **mist.utils.mpsToKnots(number** *n***)**

Returns the conversion of meters per second defined as number *n* to Knots.

mist.utils.mpsToKmph

number **mist.utils.mpsToKmph(number** *n***)**

Returns the conversion of meters per second defined as number *n* to Kilometer per hour.

mist.utils.knotsMps

number **mist.utils.knotsToMps(number** *n***)**

Returns the conversion of knots defined as number *n* to Meters per second.

mist.utils.kmphToMps

number **mist.utils.kmphToMps(number** *n***)**

Returns the conversion of kilometers per hour defined as number *n* to meters per second.

mist.utils.get2DDist

number **mist.utils.get2DDist(table** *point1,* **table** *point2***)**

Returns the distance between two *point1* and *point2* in 2D space.

mist.utils.get3DDist

number **mist.utils.get3DDist(table** *point1,* **table** *point2***)**

Returns the distance between two *point1* and *point2* in 3D space.

mist.utils.getDir

number **mist.utils.getDir(table** *vec,* **table** *point***)**

Returns the heading of a vector. If the optional point is given the heading is error corrected to account for the "flat" map projection on the map in DCS. The further away from the "center" point of the map, the larger the error.

Debug

For functions mist.debug.dump_G, mist.debug.writeData, and mist.debug.dumpDBs, you must (temporarily!) unprotect the Lua environment and enable the io and lfs libraries. You do this by commenting out the sanitizeModule calls in <DCS main directory>/Scripts/MissionScripting.lua. Just be sure to re-protect yourself later when you run missions from untrusted sources.

mist.debug.dump_G

mist.debug.dump_G (string *fileName***)**

Dumps the global environment (using mist.utils.tableShow) to Saved Games/DCS/Logs/ *fileName.*

mist.debug.writeData

mist.debug.writeData (function *fcn*, **table** *fcnVars*, **string** *fileName***)**

Writes to the file Saved Games/DCS/Logs/ *fileName* the results of function *fcn* called with the variables unpacked from the table *fcnVars*.

mist.debug.dumpDBs

mist.debug.dumpDBs()

Serializes all the tables in mist.DBs and outputs them to files in Saved Games/DCS/Logs/.

Vectors

Vector operations for Vec3.

mist.vec.add

Vec3 mist.vec.add(Vec3 *vec1,* **Vec3** *vec2***)**

Returns the vectorial addition of *vec1* + *vec2*

mist.vec.sub

Vec3 mist.vec.sub(Vec3 *vec1,* **Vec3** *vec2***)**

Returns the vectorial subtraction of *vec1* and *vec2* (i.e., *vec1 - vec2*)

mist.vec.scalar_mult

Vec3 mist.vec.scalar_mult(Vec3 *vec,* **number** *mult***)**

Returns *vec* multiplied by scalar *mult*.

mist.vec.dp

number mist.vec.dp(Vec3 *vec1,* **Vec3** *vec2***)**

Returns the dot product of *vec1* and *vec2*.

mist.vec.cp

Vec3 mist.vec.cp(Vec3 *vec1,* **Vec3** *vec2***)**

Returns the cross product of *vec1* and *vec2* (i.e., *vec1* X *vec2*)

mist.vec.mag

number mist.vec.mag(Vec3 *vec***)**

Returns the magnitude of *vec*.

Time

The mist.time functions are built around converting game time to a readable and useful format. DCS has long defined the start time of a mission in seconds instead of dates and time. A start time of 0 is always June 1, 2011 with each day adding 86400 seconds to that value.

2:00 AM June 1st in seconds = 7200
2:00 AM June 2nd in seconds = 93600

mist.time.getDate

table **mist.time.getDate(number** *time***)**

Returns a table formatted as {d = day, m = month, y = year} of a given time in seconds. If no value is passed the function returns the current date of the mission.

mist.time.getDHMS

table **mist.time.getDHMS(number** *time***)**

Returns a table formatted as {d = day, h = hours, m = minutes, s = seconds} of a given time in seconds. If no value is passed the function returns the current time of the mission.

mist.time.convertToSec

number **mist.time.convertToSec(table** *DHMSTable***)**

Returns the *DHMSTable* input value to a numerical value in seconds.

mist.time.milToGame

number **mist.time.milToGame(string** *mil***, boolean** *timeUntil***)**

Returns a numerical value in seconds of the next occurrence of the passed *military* time string. If the boolean value *timeUntil* is present the value returned is the time it will take to reach the specified military time. String military must be exactly 4 characters.

Examples. Assuming these scripts are run on day 0 at 1 AM (3600 seconds)
mist.timeMilToGame('0200') returns 7200
mist.timeMilToGame('0200', true) returns 3600

If scripts ran at 3AM (10800 seconds) -- Note script returns the NEXT occurrence
mist.timeMilToGame('0200') returns 93600
mist.timeMilToGame('0200', true) returns 90000

Logger

Mist 4.2 has added a new logger functionality to help debug code and provide more detailed information if an error occurs. The logger functions as a class and shares similar functionality to the DCS scripting logging functions, however it has its own benifits. Primarily the logger will display the line in your code where it was called at. Logs are added to your DCS.log in the following format:

timestamp type SCRIPTING: LoggerName|Line logger was called at: message here

01161.112 WARNING SCRIPTING: Bobs Script|4: This is a warning message

mist.Logger:new

object mist.Logger:new(string *name*, string/number *level*)

Creates the logger object of the specified *name* and message *level*. The level is useful in selectively changing which log types may appear. If level is not specified a value of "2" will be set. The level defines the maximum type of log that may appear. Levels can be defined as a string or a number with the following values:

0 = 'none' or 'off'
1 = 'error'
2= 'warning'
3= 'info'

If the level is set to 1 then any use of warning or info logs will be ignored.

Examples of how to create a logger:

      myLogger = mist.Logger:new("MyScript")
      myLogger = mist.Logger:new("MyScript", 2)
      myLogger = mist.Logger:new("MyScript", "info")

Each function that can display a message has the capability of replacing keywords with text relevant to the error message. Keywords are created by a '$' symbol followed by a number that defines the index associated with it.

The parameters can be created by using a table or additional input values.
For example:
myLogger:msg('Im afraid my $1 is $2', 'parrot', 'dead')
myLogger:msg('$1 I need $2 power immediately!', {'Mr Scott', 'warp'})

Member functions:

myLogger:setLevel

nothing myLogger:setLevel(string/number *level*)

Modifies the loggers current *level*. Can be used to change the level of the logger at runtime. See mist.Logger:new for valid entires for level.

myLogger:msg

nothing   myLogger:msg(string *message,* **table/multiple entries** *params***)**

Message logs will always be logged no matter the level set.

myLogger:error

nothing   myLogger:error(string *error,* **table/multiple entries** *params***)**

Error logs are prefixed with 'ERROR' in DCS.log. Will be logged as long as level is 1 or greater.

myLogger:warn

nothing   myLogger:warn(string *warning,* **table/multiple entries** *params***)**

Warning logs are prefixed with 'WARNING' in DCS.log. Will be logged as long as level is 2 or greater.

myLogger:info

nothing   myLogger:info(string *info,* **table/multiple entries** *params***)**

Info logs are prefixed with 'INFO' in DCS.log. Will be logged as long as level is 3.

myLogger:alert

nothing   myLogger:alert(string *alert,* **table/multiple entries** *params***)**

Alert logs are prefixed with 'ERROR' in DCS.log. Alert logs cannot be disabled and will create a error message window pausing the simulation.

Demos

Demonstration scripts.

mist.demos.printFlightData

number   mist.demos.printFlightData(Unit *unit***)**

This function outputs to screen significant flight parameters of the Unit-type object *unit*. These parameters are:

-Heading
-Pitch
-Roll

-Climb Angle

-Angle of Attack

-Yaw

-Angle of Attack + Yaw (they are at right angles so AoA + Yaw = $(Yaw^2 + AoA^2)^{0.5}$).

-Speed

-Absolute Acceleration

-Axial G loading

-Transverse G loading

-Absolute G loading

-Specific energy

-Specific dE/dt

This script can even be applied to weapons, so you can use it to view in real time the flight characters of something like an AMRAAM!

Databases

Mist databases are a collection of global tables consisting of a wide variety of data. Most DBs are created once at the start of the mission and are all based off of information found within the mission editor.

As of Mist v3.0 several databases will be updated if groups are dynamically added either via the built in mist functions or other scripts. Some values are not accessible to the scripting engine, like aircraft skill, at which point Mist will automatically assign a skill level of "random" if the skill is not known.

Several of the databases have overlapping information that is found in another database, this is done for convenience. New databases have been added to fill that are solely based on mission editor information, these databases will not be adjusted as groups are added. If a group new group with an already pre-existing name, it will overwrite the old group.

For specifics of how the data is organized for each DB, please see the "Example DBs" folder from the Mist download file.

Zones

mist.DBs.zonesByName

      Filtered by name of trigger zone

mist.DBs.zonesByNum

      List of zones numerically indexed

Both are a database of triggerzones as defined within the mission editor.

- name

- zoneId
- x
- y
- radius
- point in **Vec3**
- hidden in ME
- color in ME

mist.DBs.navPoints

Table of navigation points indexed by coalition.

- name
- callsignStr (callsign String)
- groupId
- x
- y
- point in **Vec3**
- properties

Units

Each entry in the unit DBs contains the following information:

- unitName
- groupName
- x
- y
- point
- coalition
- country
- category
- type
- skill
- unitId
- groupId
- countryId
- starting point in **Vec2**
- speed (applies only to aircraft)
- livery_id (applies only to aircraft)

- onboard_num (applies only to aircraft)
- callsign (applies only to aircraft)
- psi (applies only to aircraft)
- AddPropAircraft (applies only to aircraft)
- canCargo (applies only to statics)
- mass (applies only to statics)
- categoryStatic (applies only to statics)

Tables with "ME" in the name are static and will not be modified if groups are dynamically added to the mission.

mist.DBs.units
mist.DBs.MEunits
> Table of all units within the mission indexed by coalition, country, type, and groups.

mist.DBs.unitsByName
mist.DBs.MEunitsByName
> Table of all units indexed by unit name

mist.DBs.unitsById
mist.DBs.MEunitsById
> Table of all units indexed by unit Id

mist.DBs.unitsByCat
mist.DBs.MEunitsByCat
> Table of all units indexed by category.

mist.DBs.unitsByNum
mist.DBs.MEunitsByNum

> Table of all units indexed numerically starting at 1.

Groups
Each group table contains the following information
- groupName
- coalition
- country
- category
- groupId
- countryId
- uncontrolled
- hidden

- startTime
- frequency (applies only to aircraft)
- modulation (applies only to aircraft)
- radioSet (applies only to aircraft)
- unit table for each unit in group

Tables with "ME" in the name are static and will not be modified if groups are dynamically added to the mission.

mist.DBs.groupsByName
mist.DBs.MEgroupsByName
> Table of all groups indexed by groupName.

mist.DBs.groupsById
mist.DBs.MEgroupsById
> The same as mist.DBs.groupsByName, but indexed by groupId instead.

Dynamically Added

Tables of all objects added dynamically via the scripting engine. Indexed numerically.
mist.DBs.dynGroupsAdded
> Same format as Groups tables, however it contains the simulator time of when the group was spawned in addition to the mission time.

Constants

The constants table is a collection assorted data that is not directly accessible to the scripting engine but still applies to it.

mist.DBs.const.callsigns
> Table of available NATO callsigns and scripting engine enumerators related to each callsign.

Clients

mist.DBs.humansByName
DB of humanable aircraft indexed by unitName, contains all the standard unit info:
- unitName
- groupName

- coalition
- country
- category
- type
- skill
- unitId
- groupId
- countryId
- starting point in **Vec2**
- task

mist.DBs.humansById

      The same as mist.DBs.humansByName, but indexed by groupId instead.

Real time Databases

The following databases are updated automatically as the mission progresses. Both databases can share the similar sets of information. The dead objects database will also contain information for world objects which become destroyed over the course of a mission. In deadObjects, the "same" units (client aircraft) will have multiple entries if the same aircraft was spawned into the world and died more than once.

mist.DBs.aliveUnits

      A list of all units (helicopter, plane, ship, vehicle) that are currently alive, indexed by the unit's runtime id_ value. Each entry contains a copy of the data from that unit's entry in the units DBs, and also contains the unit's current position (in Vec3) at table key "pos". This DB is not instantaneous- it is refreshed between 1 and 20 times per second (the "refresh" rate will vary depending on the number of units in the mission- the more units, the slower it refreshes). Inactive units (units that are not yet activated) will be listed in aliveUnits too (because they ARE alive units).

mist.DBs.deadObjects

A list of dead objects indexed by the dead object's former runtime id_ (However, in the case of duplicate runtime ids, the index can be a string, like "11565326 #1").

There are several types of dead objects, at the table key "objectType" for each object:

      "helicopter"

"plane"

"ship"

"static" – a dead static object

"vehicle"

"building" – a dead map object.

"unknown" – type of dead object could not be determined.

For "helicopter", "plane", "ship", "static", and "vehicle", mist.DBs.deadObjects will copy in the data from that dead object's entry in the units DBs at table key "objectData".

For all objectTypes except "unknown", the last known/current position of the dead object is listed at table key "objectPos" (in **Vec3** format).

The object itself is listed at table key "object" (though, it might not be casted into an Object class object).

Miscellaneous
mist.DBs.missionData

Table of basic mission information containing:

- Mission editor version number
- Files within the .miz
- mission start time
- Name of theatre of war used
- red/blue bullseye location in **Vec2**

With DCS 1.5 the files within the miz database might not fully populate the files due to changes in .miz file.