Multiplayer Mission Making For Arma 3

Contents

In recent weeks and months I have come across a couple of missions while playing with our squad that were either not functioning correctly or were downright broken. A lot of the issues can be explained by the simple fact that we play on a dedicated server as opposed to hosted games. In this document I will try to outline my experience with mission making and how to make solid, working missions that are compatible to both single- and multiplayer, as well as working on hosted and dedicated environments.

My main focus is no-respawn, hardcore coop, and that is what I will be focusing on. A lot of the multiplayer concepts in Arma 3 are, however, applicable to all types of missions. However, large-scale PvP scenarios will have other performance and optimization issues than low- to mid-scale coop missions. My experience is with the latter, specifically, missions with up to 20 or so players against AI.

This is not intended to be an all-out scripting guide. A certain proficiency with Arma 3 Scripting in general is assumed for this. We'll only concentrate on the multiplayer parts, and there mainly on the pitfalls people might encounter.

Locality – The Most Important Thing

One of the fundamental mission maker tools is a knowledge of locality. In Arma 3, we do not have a strict client-server architecture. Instead, all clients as well as the server itself run code, both mission-specific user code as well as engine-internal AI code. The concept of Locality means nothing else but to determine where an engine object or entity is currently stored, and how this

affects the execution of commands. Likewise, it is important to know how and where global variables are stored. We will examine these things in detail below.

Locality and Variables

The locality mostly concerns objects, but other things need consideration too. For example, Arma differentiates between global and local variables, but the names are a bit misleading. Local variables are local to a certain scope, namely the scope they are defined in. They are invariably named with a leading underscore character. The scripting engine immediately forgets them if they go out of scope, which is a common pitfall if you define them e.g. in the “then” part of an if statement.

What Arma calls global variables are variables that start with something else than an underscore character. However, and this is very important, global variables are global only to the scope of scripts running on the machine they were defined on. What this means is that if my script runs on a client machine and assigns a value to a global variable, that value is only visible on this specific machine. Even worse, a variable with the same name can have a different value on a different client, or the server. Arma does NOT automatically distribute variable values around clients and the server.

In this respect, we refer to “clients” as those machines that run a player's game, and “server” as the machine that hosts the game. On a hosted game, one client will be identical to the server, but on a dedicated host, this is not the case.

In order to make sure that a variable has the same value on all clients and the server, it needs to be declared as public using the `publicVariable` command. In a script this looks something like this:

```
/* Define a variable to be used on all attached clients */
FHQ_DifficultyLevel = 1;
publicVariable “FHQ_DifficultyLevel”;
```

The above makes sure the variable has the same value on all clients and the server, but this works on a first-come-first-served basis, meaning that running the same command on more than one client with different values might have different outcomes.

It is also worth nothing that although the variable in the above example was declared public, this **only** applies to the current value of the variable. This does **not** mean that future changes of the variable will be automatically transmitted as well. Every time you change the variable and want that change to be visible outside of the current client or server, you need to run the `publicVariable` command again.

This type of locality behaviour also affects variables set with the `setVariable` command. This command is usually used on objects or namespace objects, and the default behavior is the same as global variables – content or changes is **not** propagated to other machines. In order to make this type of variable public in the same sense as above, you need to use the three-element array version:

```
/* Make an object's variable public */
MyChopper setVariable [“Callsign”, “Zulu Victor Niner”, true];
```

The highlighted parameter is a boolean that will determine whether this `setVariable` command is propagated or not. Obviously, setting it to `true` will propagate the value, while omitting the parameter or setting it to `false` will not.

What you should have gotten out of this section:

1. “global” variables are local to the machine they are defined on. Their value or changes in their value are not automatically known on other machines.
2. To make a variable really global, you need to use the `publicVariable` command every time you changed a value.
3. These rules apply to object specific variables as well, you need to use the three element array version of `setVariable` to ensure distribution over the network.

Locality and Objects

Unlike variables, objects always reside on exactly one machine. This can be the server or any connected client, and it can and will change with time (with certain exceptions). The exact rules of locality are outlined here, but as mission makers it isn't really all that relevant to us. What is important, though, is to know when and where effects are global or local, and what type of arguments a scripting command can take. To test whether an object is local or not, the command `local` *`object`* can be used which will either return `true` if the object is local to the machine the command is running on, or `false` if it isn't. This is a very important command.

For the sake of clarity, when we say object here we refer to a vehicle, a unit, or anything that is of type `Object`. A common predefined object variable is `player`, which is a reference to the object that represents the player on the client the script is running on[1]. On a client, the scripting command `local player` will always return `true`, and consequently, this means that the player variable references a different object on each client (and no object on the dedicated server).

If you look at the BIKI entries for some commands (most notably, those that refer to objects), you will notice some two of four different icons that can appear on top. These are:

1. “AL” (argument local): A command that has this icon needs to be run on the machine that the object it operates on is local. That means that if the object is on a different machine, the statement will have either no effect, or a different, undefined effect. `setVariable` is an example of a command with a local argument; the statement will have no effect if the argument is on a different machine.
2. “AG” (argument global): A command with this icon can run on any machine, the argument object does not need to be local (but can be) to the machine running on. An example of this is the `local` command, which can operate on any object.
   AG is mutually exclusive to AL
3. “EL” (effect local): A command with local effect will have exactly that – a local effect. The result of this command will be local to the machine it runs on, and its effects will not be visible globally, on different machines. Again, the `setVariable` command is an example

1 Note that there is no player object on a dedicated server, only on clients (or a hosted server), a common pitfall that is responsible for a lot of errors.

of a command with local effect; unless instructed to do so by the three-element array version, `setVariable`'s effect will only be visible locally, and not propagate over the network.

4. “EG” (effect global): A command with global effect will have a visible, noticeable effect on all connected machines (clients and server alike). The `setDamage` command is one such example, even executed on a single machine, setting an objects damage to 1 will kill it and it will appear dead/destroyed on all attached machines.
   EG is mutually exclusive to EL.

It is of utmost importance to know what argument type and effect type a command has. Any mistake here has the potential of breaking your mission, so make sure to look up the effects.

Object locality will dictate what kind of scripting commands you can run on an object. If `local` *`object`* returns `false`, then you will not be able to use a command of the “AL” category. This category of commands **must** run on the same machine. On the other hand, a command of the “AG” category can disregard the locality of the object, since it will be able to operate on it (however, the effect can still be local only).

A command with local effect cannot be easily made global. In the case of the `setVariable` command, there is an easy way to distribute the effect to other machines, but that is not the case for all commands. As an example the `addWeaponCargo` command, while being able to operate on an object that is stored anywhere, will only add the weapon cargo to the object on the specific machine that the command runs on. So, for example, if you run the command on an ammo box on a dedicated server, the extra weapons you put in are **not** visible to the clients, and it will appear as a stock ammo box.

Luckily, the `addWeaponCargoGlobal` command is a version of the command that *will* do just that, so in this case it is easy to work around this limitation. Other commands might not be so easy to fix, but we will talk about that later in the discussion of the `BIS_fnc_MP` function.

Common Pitfalls with Global Effect commands

It is very important to observe that the global effect takes place every time the command is executed. Take the following example of an `init.sqf` script from a fictuous mission. We will assume there is a car in the mission called “FHQ_playerCar” placed on the map, and you want to fill it up with four MX rifles and 10 magazines.

```
/* init.sqf */
FHQ_playerCar addWeaponCargoGlobal [“arifle_MX_F”,4];
FHQ_playerCar addMagazineCargoGlobal [“30Rnd_65x39_caseless_mag”, 10];
```

If you paid attention, you will immediately see what is wrong here. `init.sqf` is run at startup on all clients and the server. The two lines of code supposedly add 4 MX rifles and 10 magazines. But since the effect is global, something else entirely will happen.

- On a hosted game with only a single player, or single player for the matter, the effect will be as intended – there will be 4 rifles and 10 magazines.

- On a dedicated server with a single player, there will be 8 rifles and 20 magazines. Since the script runs on all machines, and the effect of the command is global, `addWeaponCargoGlobal` will add four rifles on all machines. In fact, if there are more players than one, the effect will stack.

This is just a very constructed example, but it shows the problem with global effects: Duplication. Not even Bohemia Interactive's own code is always bulletproof against that. For example, the Supports module that airdrops ammo will duplicate by the number of players in a mission on dedicated servers.

Similar effects can occur when using triggers. We will talk about trigger locality in more detail below.

Common Pitfalls with Local Effect commands

Obviously, one of the most serious issues that can occur with local effect commands is that they only have effect on a single machine. `createSimpleTask` is an example of a command that takes a global argument but only has local effect. While most briefings are added during `init.sqf` and therefore globally visible, creating tasks dynamically during the mission will fail to show the tasks for everyone unless the command is properly executed on all attached clients.

Likewise, certain commands that seem to be working correctly when you test the mission in single player or on a hosted environment will fail to work on dedicated or with more players. For instance, if you want to override an object's material, you would use the `setObjectMaterial` command. Typically, you would be tempted to put this into e.g. a vehicle's init field. However, the init field is run only where the object is local – if it is an empty vehicle, it would only run on the server, if a player is sitting in it, it would only run on the machine that the driver is using, etc. Since the effect is local, at most one player would see the effect, and everybody else would not see anything.

What you should have gotten out of this section:

1. Script commands that operate on objects may require a local argument. In this case, supplying a non-local argument might not have any effect at all.
2. Script commands might have local or global effect. Local effects are only visible on the machine running the script, while global effects take place everywhere.
3. Objects in Arma have one and exactly one machine were they are stored (are “local”). The owner can be any client or the server, which might be the same on a hosted server.
4. Common problems include duplication of effects (when a global effect command is run on multiple machines) and effects only visible on at most one machine (when a local effect command is only run on a specific machine).
5. The problems mentioned in the above point might or might not be visible on a hosted test server if you only connect with a single client.

Locality of Triggers

Triggers created either in the editor or using the `createTrigger` command are always global objects. This means they should only be created on the server, and not necessarily on the client (although I am not sure about radio triggers, since you might want to give only one side a radio trigger and not the other). Running `local triggername` on a client that is not the server (i.e. a joined player or any client on a dedicated server) will return `false` for triggers that have been placed in the editor. In this respect, triggers are to be treated like any object in terms of locality, so that e.g. the command `setTriggerActivation` should run on the server since its effect is local.

When we speak about locality of triggers, we are more concerned with how they execute their statements or create their effects. Unfortunately, this isn't as clear cut as it seems.

A radio trigger (for example Radio Alpha) will very likely run its statements on every client and the server when triggered. That means that the scripting commands in the activation field will run for at least the number of players connected, potentially one more for the server. This means that the duplication effect from global effect commands discussed in the previous chapter is very obviously a danger here.

However, as a general rule, trigger activation statements (and therefore, triggering the trigger) ***only*** run on the machine where the trigger condition evaluates to true. Let me repeat this again: The trigger only fires on those machines were the condition evaluates to true. This very important fact can have unforeseeable side effects. A common example is the “Detected By” trigger. It is very possible that a player is detected on one or more clients and/or the server, but not on others. If you use such a trigger to end a mission (say, a stealth mission which requires users to stay undetected), it might end up terminating the mission on some clients but not on others.

Another example are triggers that have a scripted condition. For example, putting `!alive player` into the condition of a trigger will only trigger on the clients were players have been killed. While this effect is obviously desirable, combined with other locality issues it is a very common source of problems.

Getting a grip on these issues isn't necessarily difficult, but one needs to be aware of it.

What you should have gotten out of this section:

1. Triggers behave like normal objects in terms of locality, but should, as a rule, only be created on the server.
2. Most trigger conditions are globally valid, but the word “most” is the problem here.
3. Triggers only trigger on machines that satisfy their trigger condition. Most notably, using scripted trigger condition other than `this` will be highly dependent on locality and variables.

Practical Tips and Tricks

Armed with the knowledge above, the rest of this guide will try to give advice on how to handle common situations. This is by no means complete, and some of it might not be the optimal solution,

but so far, it has (mostly) worked for me.

Briefings And Tasks

As mentioned above, a good number of scripting commands related to briefings and tasks are local effect only. Obviously this is a necessity, since you do not always want all players/groups/sides to have the same briefing. However, this means that getting a briefing that works reliable will have its challenges.

Luckily, there is a very easy solution to this issue: Use a task tracker. I am not trying to promote FHQ Task Tracker here, use whatever you like. FHQ and Shuko's are two example of working task trackers that will make the creation, both statical and dynamical, of briefings and tasks so much easier. Both of them are robust and proven to work. Really, there is no reason whatsoever to reinvent the wheel – use a task tracker for briefings, and save yourself a lot of headaches.

Working With Triggers

As we have seen above, triggers might be tricky to handle correctly, especially if you do not want them to behave erratically like the *Detected By* type. There are a couple of things you can do. One method I had frequently used in the past is Game Logic objects and waypoints. If you synchronize a trigger to a waypoint, the group associated with the waypoint will move to it but do not continue until the trigger fires. This can be used to e.g. run a script or commands in the waypoints completion field. However, I do advice against this method, or rather, I don't use it anymore since I have found that (at least last time I checked) logic waypoints (AND and OR types) are broken (AND waypoints behaved like OR waypoints).

What I mostly do myself these days is to use a mission flow FSM. FSM stands for Finite State Machine, and what it means is a graph of nodes that can either be states or conditions. When a state is reached, the FSM execution will halt in that state until one of the attached conditions evaluates to true. For more on FSM's, see the chapter “FSM Crash Course” below.

The major point with triggers is making them reliable and determined, meaning that you want to make sure that the trigger works and its statements are only executed the right amount of times (mostly, one time, or one time per client). For this purpose, any method works if it first determines that the trigger fired on a machine, and then executes the correct statements. Using the above waypoint method will make the script execution only depend on the game logic (which only exists once, and always on the server) and therefore will fold one or many firings of the trigger to a single one which can then continue the processing in a deterministic way – you will be absolutely sure that the code will only be executed once, so you can use whatever methods to make sure it runs were it is supposed to run.

Triggers can be used effectively using a condition field containing a variable. For example, to end a mission and make sure that it runs on all clients, place a trigger in the mission and put the following in the Trigger Condition:

```
myVariable
```

In the On Activation field, place the following:

```
[“end1”] call BIS_fnc_endMission;
```

Now, to reliably end the mission, any client or the server can simply run

```
myVariable = true; publicVariable “myVariable”;
```

Obviously, you can pick any variable name, in fact, you are encouraged to use a sensible one. You should also set the variable to `false` in your `init.sqf` to make sure it is defined. Triggers like this are very easy to make and generally work well.

Using `BIS_fnc_MP`

For most missions, it isn't obvious how to run something on a specific machine in a multiplayer environment. The standard function set of Bohemia Interactive's function module, however, contains a function that is tailored to handle this specific issue: `BIS_fnc_MP`.

The basic premise of the function is to run a piece of code (mostly a function, i.e. a precompiled variable containing code) on a number of machines. In its simplest form, it takes the arguments and the name of the function (or a command name) and runs it on all clients and the server:

```
[“Hello World”, “systemChat”, true] call BIS_fnc_MP;
```

The first parameter is the argument to the remotely executed command or function. In this case, since the `systemChat` command accepts a single string, it is just that. More complex examples will likely have an array here.

The second parameter is the command or function to run. If it is a function, it must exist as a variable on the remote machine(s). The `systemChat` is a command that displays a message in grey (running `systemChat` on the server of course doesn't make sense, since the server doesn't have a display).

The third parameter may vary wildly depending on what you want to do. In this case, it is a boolean, which indicates that you want to run the command on the server and (since it is `true`) on all clients. Were it `false`, you would only run on the server.

There are other possibilities for this parameter. For example, specifying an object will run the code only were the object is local and nowhere else. Likewise, you can specify a side or a group to run it only on clients where the player is on that specific side, or in that specific group.

There are more options to this function, check the Wiki page for more details.

Ending A Mission

Ending a mission needs to be done on all machines, not only one. Missions that adhere to the new mission presentation guidelines put out by Bohemia Interactive should end the mission by calling `BIS_fnc_endMission`, but again, this needs to be called on all machines. This can easily be done using any of the above methods (`BIS_fnc_MP` or Triggers), the reason I make specific mention here is that it needs to be done and is often forgotten.

A FSM Crash Course

What is a FSM

Note: In order to use FSMs, you need the Poseidon Tools installed. You can find more information

here. More specifically, you need the program “FSMEdit”.

FSM stands for “Finite State Machine”, and in the traditional sense is a directed graph with nodes representing states and labelled edges representing transitions from one state to another when reading an input character. Bohemia's FSMs are a slight bit different, but the concept is the same. An FSM in Arma is a graph consisting of two basic types of nodes: States and Conditions.

States are like the states in a traditional FSM. They contain some code, and can contain any number of links to other nodes, but the other nodes **must** be Conditions. There are three basic State types: Start state, End state, and User state.

User states are the standard states that are used within the FSM (they are displayed as white boxes by default). Typically, almost all states in an FSM are user states. There can only be one Start state. The Start state is like any other user state, but this is where execution of the FSM begins (it is shown as a red, rounded rectangle by default). Finally, there might be any number of end states, and they too behave mostly like any other user state, only that once their code is executed, instead of evaluating the conditions (see below), the execution of the FSM terminates (end states display as an orange rounded rectangle).

Conditions, on the other hand, are nodes that evaluate a condition (hence the name). If the condition evaluates to true, then the FSM's state changes to the one (and exactly one) node that the Condition is linked to. Two basic conditions exist that are interesting for our purposes. A Condition is what is normally used, it is displayed as a yellow diamond shape (like the conditions in flow charts). A condition has a code field that is expected to yield a boolean result. On the other hand, a True Condition (represented by a greenish diamond) is a Condition that always returns true. It is somewhat redundant since it could be simulated by adding a “true” to a normal condition's code field.

FSM Execution

Execution starts at the Start node. Every time a state node is entered, it first executes its “InitCode” code, which in our case will be a simple SQF script snippet. This works like calling a code piece in a normal SQF script.

Once the code has run, the FSM interpreter goes through all linked conditions of the current state, evaluating their Condition field. The first one to yield a true result is taken, and the FSM transitions to the state that the Condition links to. This process is repeated until an end state is reached or hell freezes over (well, actually, the mission ends/the server terminates, but for all intent and purpose, the FSM will run forever unless terminated).

The order that the conditions are evaluated is largely first-come-first-served. However, each of the Condition nodes can have a priority (which defaults to zero). The higher the priority, the earlier the condition is considered.

If the new state is an End state, the code of that state is executed normally, but execution of the FSM terminates.

It is important to note that if no condition evaluates to true, the FSM stays in the current state but the code is not executed again. Code is only executed one the state is reached, either for the first time or after the FSM performed a loop and execution returned to a specific state. On the other

hand, the code of Condition is executed every time the condition is considered.

Convoy Ambush Example

Let's start to create a small FSM for a fictitious mission. Let's assume we have a convoy consisting of two vehicles (named eastVehicle1 and eastVehicle2 in the editor). We want to have the players ambush and destroy the convoy, then fall back to an LZ were they will be picked up by a helicopter (named westHelo1) and flown out.

If you are a seasoned mission maker, you can just read the following to get what we're talking about. If you are unsure, I recommend you follow along with this. I will require a basic knowledge of mission making, but I guess most of it is pretty obvious.

Multiplayer missions are best edited in the MP Editor. Start a hosted MP session and choose “New Mission” on Stratis. We'll assume our convoy is coming from Mike-26 headed for Rogain, and that we want to ambush them in the small forest area around 048044.

Start with the player team. I have chosen to make it a four player mission, with a Recon team leader, AT specialist, Demo expert and Paramedic. My starting point is just north of the proposed ambush position, so players have time to move there and set up. Pick a date, weather, and time of day. I will make it early morning and slightly rainy.

Place a marker on the road, name it “markAmbush” and give it “Ambush Convoy” as the text. You may want to rotate it so that it points down the road towards the direction that the convoy is coming from. Likewise, place a marker at 053037 and call it “markPickup” with a text of “Extraction”.

Let's take two T-100 tanks, put them on the road near Mike-26, and call them eastVehicle1 and eastVehicle2. Place a first waypoint right in front of them and set it to “Safe” and “Column”, then continue to set waypoints until they arrive at their final destination somewhere near Rogain (you can put it closer to the ambush point so that failure of the mission will not be drawn out too long, but give the players enough time to pursue the tanks and engage them with AT if needed).

Now place a trigger near the first waypoint, and set its Condition field to true. Set the timer to “Countdown” and set Min, Mid and Max all to 60. Synchronize the trigger with the first waypoint. This will ensure that the convoy waits a minute before departing Mike-26.

We want to know when they arrive there, so that we can fail the mission when this happens. In the last waypoints On Activation field, enter the following:

```
ConvoyReachedDesitnation = true; publicVariable “ConvoyReachedDestination”;
```

This will, when the final waypoint is reached, set a global variable to true. Save the mission, then create a file init.sqf in the mission folder, and place this in the file:

```
ConvoyReachedDesitnation = false;
tf_no_auto_long_range_radio = false; // No long range radio required (TFAR)
call compile preprocessFileLineNumbers "fhqtt2.sqf";
call compile preprocessFileLineNumbers "briefing.sqf";
```

The second line makes sure we do not get long range radios on the team leader if TFAR is loaded; we don't need it for this mission, and would rather have him have a backpack with extra explosives (see below). Since this is executed on all machines, there is no need to make it a

`publicVariable`. The second line initializes the FHQ Task Tracker that we are going to use for briefings and tasks. Download FHQ Task Tracker from here and copy fhqtt2.sqf into your mission folder.

Now create a new file `briefing.sqf` and copy/paste the following code in.

```
/* Define the briefing */
[
      west,
            ["Mission",
                  "Ambush the convoy on the <marker
name=""markAmbush"">forest road</marker>"
            ],
            ["Execution",
                  "A convoy of two T-100 tanks is leaving Mike-26 in a few
minutes. Move your team to the road through the forest at <marker
name=""markAmbush"">048044</marker>, prepare an ambush, and make sure that
the tanks never reach Rogain.<br/><br/>After the ambush, fall back to
<marker name=""markPickup"">053037</marker> for extraction by helicopter."
            ]
] call FHQ_TT_addBriefing;

[
      west,
            [
                  "taskAmbush",
            "Proceed to the <marker name=""markAmbush"">forest road</marker>
and set an ambush for the two T-100 tanks coming from Mike-26. Make sure
both tanks are destroyed",
                  "Ambush Tanks",
                  "AMBUSH",
                  getMarkerPos "markAmbush", "assigned"
        ],
            [
                  "taskExtract",
                  "Move to <marker name=""markPickup"">053037</marker> for
extraction by helicopter",
                  "Extraction",
                  "MOVE",
                  getMarkerPos "markPickup"
            ]
] call FHQ_TT_addTasks;
```

If you test-run the mission now, you will notice that you have a briefing and tasks. You will also notice that the equipment the explosive specialist carries is a wee bit underpowered for the purpose, so let's pack him some lunch.

Go to the Explosive Specialist and give him a name (I chose westMan3, since I named the others westManX accordingly). Then, add the following to the end of the init.sqf:

```
/* Make sure we have the means to destroy two tanks */
if (isnil {westMan3 getVariable "loadout"} && local westMan3) then {
    removeBackpack westMan3;
    westMan3 addBackpack "B_Carryall_cbr";
    _backpack = unitBackPack westMan3;
    clearMagazineCargoGlobal _backpack;
    clearWeaponCargoGlobal _backpack;
    _backpack addMagazineCargoGlobal ["SatchelCharge_Remote_Mag", 2];
    _backpack addMagazineCargoGlobal ["ATMine_Range_Mag", 1];
```

```
    _backpack addItemCargoGlobal ["ToolKit", 1];
    westMan3 setVariable ["loadout", 1, true];
};
```

We do a similar thing to the team leader to ensure we have enough explosive power. His variable name is westMan1:

```
if (isnil {westMan1 getVariable "loadout"} && local westMan1) then {
    removeBackpack westMan1;
    westMan1 addBackpack "B_Carryall_cbr";
    _backpack = unitBackPack westMan1;
    clearMagazineCargoGlobal _backpack;
    clearWeaponCargoGlobal _backpack;
    _backpack addMagazineCargoGlobal ["ATMine_Range_Mag", 3];
    westMan1 setVariable ["loadout", 1, true];
};
```

You might wonder what the term `isnil {westMan3 getVariable "loadout"}` is good for. The answer is simple: Without this, every time a player joins in progress into either the team leader or explosive specialist slots, it would go through the `init.sqf` and add the loadout again. Since we don't want that, we need to make sure that we “mark” our units so that `init.sqf` can determine that they got their loadout. Obviously, we cannot do that in `init.sqf`. The easiest way is to check if the unit has a variable `loadout` set, and if it doesn't, apply the loadout and set the variable. This way, next time init.sqf runs for any reason, the `loadout` variable is not `nil`, and the script will not add the backpacks again.

Now, even if you manage to destroy the tanks, nothing happens. It is time to start our FSM.

In order to start from scratch, you need to perform the following steps:

1. Start FSMEdit. You will find it in the Poseidon Tools folder.
2. Set the name of the FSM in the menu FSMAttributes->FSM Name. The name must be the same as the file base name. In my case, I call my file flow.fsm and the FsmName is set to “flow”.
3. Set the compile config via menu FSMAttributes->Compile Config. The one you need is “Data\Packages\Bin\fsmEditor\scriptedFsm.cfg” in the Poseidon Tools installation folder.

You will now see a red rounded rectangle on the screen. Double click it and type “Start”, then click somewhere to deselect it. This is our starting state. We don't have any code to execute here, but if there were initializations needed, they could go here. For example, we could have assigned the loadouts in the start state, too, which would have saved us from doing the loadout variable magic, but for demonstrational purposes, I did it like this to draw attention to the JIP problem.

Make sure that the “Draw Link” icon (the third in the toolbar) is selected, click and hold the mouse on the red rectangle, and draw out a line. When you release the button, you will see an arrow leading away from the red box towards a yellow diamond box. This is a Condition. Although it isn't needed in our example, we will now add a branch to the FSM that exists if we are not running on the server. Repeat the Draw Link action from the yellow diamond to get a white box.

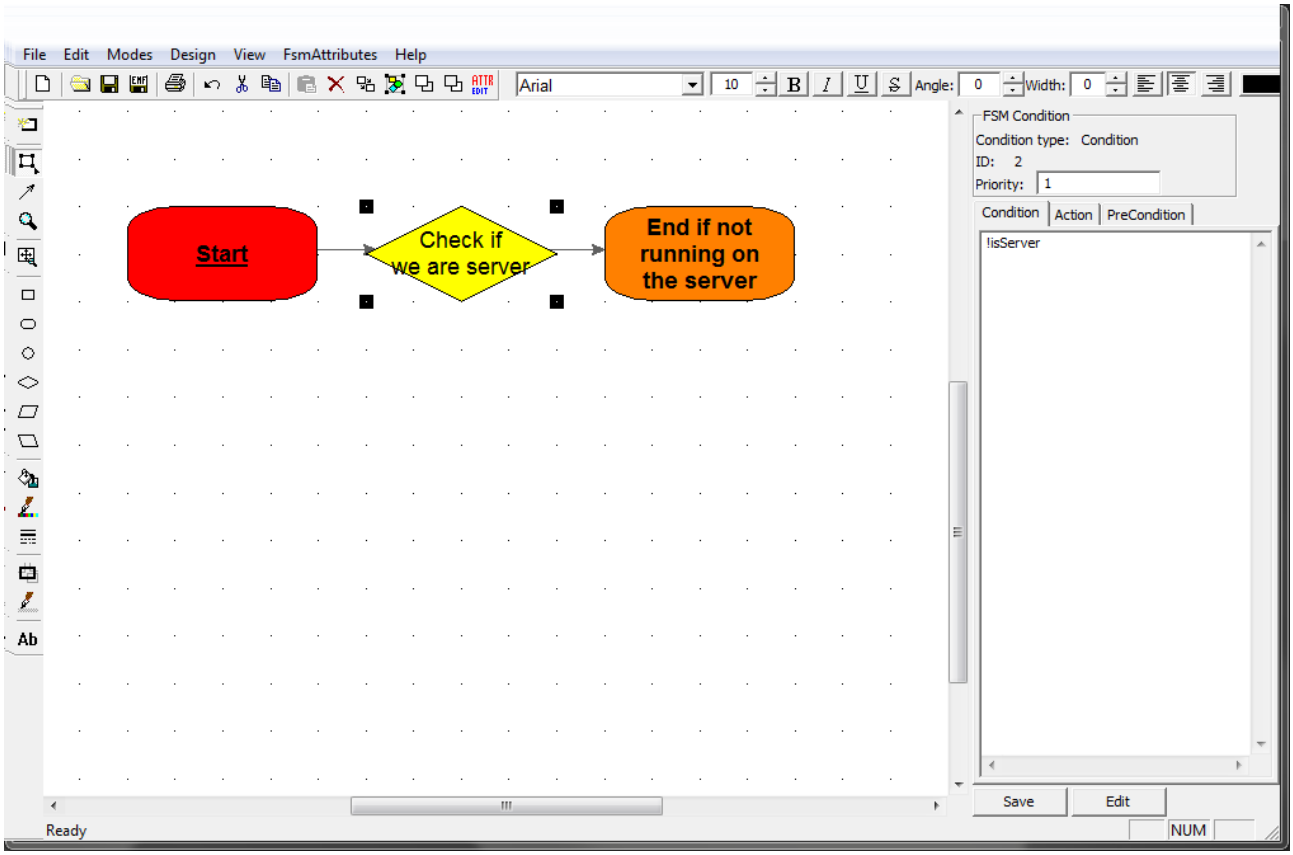
Double click the yellow diamond and enter “Exit if Server”. This is a purely descriptive text and

doesn't change the functionality, but it is a good idea to keep your states and conditions labeled. In the priority field, enter a 1. This means this condition is always checked before the default of 0. Now in the Condition field, enter `!isServer`.

What you have just done is define a potential exit for the starting state. The code "!isServer" is evaluated and will return true if we are on a client. So, on all clients that this FSM is run on, this exist is taken and will lead into the still white box.

Right-click on the box, and from the popup select "End State". The box will turn orange and get a rounded rectangle outline. This means this state is now an end state, and getting here will terminate the FSM. Enter a description like you would for any other state (like "End the FSM if not on a server").

Note that when the "Set mode to Design" button is pressed (the second icon from the top) you can click and drag the nodes around. Tidy them up to look like this:



Save the FSM in your mission folder under flow.fsm. Make sure to set the type to "Compiled FSM" in the save dialog.

Our FSM still doesn't do much, and most of all, it isn't even running. Add the following lines at the end of init.sqf:

```
if (isServer) then
{
	FHQ_missionFSM = [] execFSM "flow.fsm";
	publicVariable "FHQ_missionFSM";
};
```

This will execute the FSM and assign its handle to the FHQ_missionFSM variable. You can pass parameters to the FSM that are handled just like they are in scripts (as a variable `_this`), but we don't need this here.

Now we want to make things interesting. We'll add an exit to the start state that will get us into an “on-mission” state. While this is strictly speaking not necessary, it introduces for us a new node (the True Condition).

Make sure “Draw Link” is active, and pull a line down from the red start state. Release the button to create a yellow diamond, then right click on it and select “True Condition”. The diamond will turn green. Now pull a link out of that downwards to get a white box (a user state). Label the new state “On Mission”.

There are two conditions that we need to check when we are “on mission”:

1. The successful destruction of the convoy
2. Whether the convoy got away

The first condition will result in a “task completed” followed by assigning the “exfiltrate” task. The second condition will end the mission.

Let's start with the second condition, since it is easier to realize. We have set a variable in the final waypoint of the tanks that gets set to true once the tanks reach their destination. We can simply check for this. With “Draw Link” selected, pull a line out of the “On Mission” state and get a Condition state. Label it “Test Failure”. Pull out a link to a new user state, right click it and make it an end state.

Now, with the previously created “Test Failure” state, just enter `ConvoyReachedDesitnation` as the condition. This variable will automatically evaluate to false (if the convoy is on the way or destroyed), but will be true when it has reached its destination.

For the end state, set the label to “Fail Mission”. For the code, we need to do the following things:

1. Set the tasks to “failed”.
2. End the mission

Copy and paste the following code into the InitCode field of the “Fail Mission” State:

```
[] spawn {
      ["taskAmbush", "failed"] call FHQ_TT_setTaskState;
      sleep 1;
      ["taskExtract", "failed"] call FHQ_TT_setTaskState;
      sleep 20;
      [["Failed", false], "BIS_fnc_endMission", true] call BIS_fnc_MP;
};
```

So what is happening here? First of all, we use a `spawn` statement to run the code. This is because we use a `sleep` command, and you cannot use any sort of delay or sleep command in an FSM. Why the sleep? Well, for one thing, we want the task hints to appear on screen to let the players know that something happened (which is what the first two lines inside the spawned code do, that's part of the FHQ task tracker) so we must allow a bit of time for them to appear. Secondly, it's going

to look weird if the mission just fails, so it is a good idea to allow a bit of time.

The final line ends the mission. However, we need to do something else for it. The “Failed” string is a debriefing entry we want to show after the mission is over. For that, we need to create a description.ext file.

In your mission folder, create a new text file and call it “description.ext”. Copy and paste the following code in there:

```
respawn = "SIDE";

class Header
{
	gameType = COOP;
	minPlayers = 1;
	maxPlayers = 4;
};

author="Alwarren"; // Obviously, use your name here :)
OnLoadName = "Convoy Ambush Example";
OnLoadMission = "Ambush the Convoy";

class CfgDebriefing
{
	class Success
	{
		title = "Mission Successful";
		subtitle = "Convoy Destroyed";
		description = "Congratulation, men. You have successfully destroyed
the convoy and extracted from the area. Well done.";
		picture = "b_inf";
		pictureColor[] = {0.0,0.3,0.6,1};
	};
	class Failed : Success
	{
		picture = "KIA";
		pictureColor[] = {0.6,0.1,0.2,1};
		title = "Mission Failed";
		subtitle = "";
		description = You failed to stop the convoy and extract from the
area.";
	};
 };
```

NOTE: description.ext is only loaded when you open the mission. That means if you have edited it, you need to leave the editor back to the mission selection screen and start it again, or the changes will not be visible. Note also that any syntax error in the description.ext will cause the game to crash to desktop, so make sure you double check those brackets and those semicolons.

If you start the mission now and wait until the convoy has reached its last waypoint, it will cancel both objectives and fail.

Now we will add the success branch, or more specifically, the condition that checks if both tanks are immobilized. Going for immobilization is generally better than complete destruction, although both are valid options.

Again, make sure you are on the Draw Links mode, then pull out a new link from the “On Mission”

state, and immediately pull out another one from the new condition. Label the new condition “Convoy down?”, and the new user state “Head For Extraction”.

The condition for the convoy destruction is easy. We just need to test whether both vehicles are unable to move:

```
!canMove eastVehicle1 && !canMove eastVehicle2
```

We could also add a `canFire` to make sure they are neutralized, or we could go all-out and use `alive` instead of `canMove` to ensure their physical destruction. It all depends on what we want to achieve.

Next, for the “Head for Extraction” state, we want to simply do two things: Set the “destroyed” task to completed, and assign the extraction task. The code for this cannot be easier, thanks to FHQ Task Tracker:

```
["taskAmbush", "succeeded", "taskExtract"] call FHQ_TT_markTaskAndNext;
```

The function FHQ_TT_markTAskAndNext will set the first task to the given state (succeeded in this case) then chose the first of the following tasks that is not yet completed as a new task. Since we only have two, this is a rather easy choice.

With the main task out of the way, we now need to check when to pick up our players. For that, we need an extra trigger at the site of extraction. Go to the editor, create a trigger, then group it with the player group. You will notice new options for activation appearing after the trigger is grouped. Set the Activation to “Whole Group”, and give the trigger a name. I chose `FHQ_extractionTrigger`.

Make sure that the trigger is placed roughly at the extraction site, and make it big enough (around 80 meters or so) so that the players don't need to be too much on the exact spot (it is kind of an immersion killer if you are trying to find the right spot to stand on just to trigger the extraction).

With the name of the trigger, we can now augment our FSM to check when the group is in the trigger. This is simply done by the triggerActivated function. Pull a new link out of the “Head to Extraction” state, and enter the following in the newly created Condition that you should label something like “Reached Extraction?”:

```
triggerActivated FHQ_extractionTrigger
```

Pull a new link and name the new state “Handle Extraction”. We need to set up a few things in the editor too. For this example, we place the chopper on the map to keep the scripting effort down, normally you would probably want to spawn a chopper for the purpose.

First, put down an invisible helipad (Under Empty → Objects (Signs) → Helipad (Invisible)) at the beach of the extraction site. Place a helicopter you want to use for pickup (like the MH-9) somewhere on the lower end of the map. Name it “westHelo1”, then place three waypoints to make it fly in a holding pattern, with the last waypoint being of type “cycle”. In the OnAct field of the first waypoint, place the statement “`westHelo1 setFuel 1;`” to make sure our chopper doesn't run out of fuel if the mission takes too long.

Now, place a new waypoint AFTER the cycle waypoint. Place a trigger next to it, and in the

condition field, replace the "`this`" with "`ChopperGoExtract`". Set the Type to "switch".

Open up init.sqf and add the line "`ChopperGoExtract = false;`" somewhere at the top of the file. This will ensure that the trigger does not fire immediately. Back in the editor, synchronize the trigger with the new waypoint you created after the Cycle.

So what did you just do? A trigger of type "Switch" can be used to force a group or unit to pick the specific waypoint that it is synchronized to. In this case, it causes our chopper to break out of the holding pattern and go for the new waypoint, thus initiating the extraction once the variable "`ChopperGoExtract`" is set to true. This is exactly what we are going to do in our FSM in the "Handle Extraction" user state. Just enter "`ChopperGoExtract = true;`" in the InitCode field. We don't need to use `publicVariable` since the chopper is AI controlled and hence on the server, and the same applies to placed triggers in the editor. This will make the chopper go for the new waypoint.

Place another waypoint closer to the extraction, and finally one directly above the extraction helipad. In this last waypoint, put the Speed to "limited", and enter the following line in the OnAct field:

```
westHelo1 land "GET IN";
```

Now, place another waypoint close to the helipad with Speed set to "Normal", and add the following code:

On the Condition field, enter

```
{_x in westHelo1} count (playableUnits + switchableUnits) == {alive _x}
count (playableUnits + switchableUnits)
```

On the OnAct, enter

```
FHQ_missionFSM setFSMVariable ["_extractComplete", 1];
```

First, the OnAct introduces yet another method of synchronizing with the FSM. It sets an FSM variable directly on the FSM to indicate when the extraction is finished. We're going to extend the FSM in a minue.

The Condition field may look daunting, but it really is quite simple. You will see that it is a test for equality. On both sides of the operator, we use the `count` command to count something. The `count` command usually counts the elements in an array, but can be extended to accept code that tests each element of the array on whether it should be counted or not.

The array we count here is `(playableUnits + switchableUnits)`. This is a shortcut to get all units that can be played, either in singleplayer or multiplayer. In single player, `playableUnits` is an empty array with `switchableUnits` contains all units that the player can switch to/take over. In multiplayer, this is the other way around. Simply adding the two will always produce the same array.

The first condition we check is `{_x in westHelo1}`. This will simply check each element of the array (i.e. all of our playable units) if they are in the chopper. Likewise, the condition `{alive _x}` will count how many of them are still alive. So essentially, we activate this waypoint if and only if

all alive members of the group have boarded the helicopter.

Now, for the final piece. Back in FSMedit, pull out one more condition and label it “Wait for Extraction”. For the condition code, use

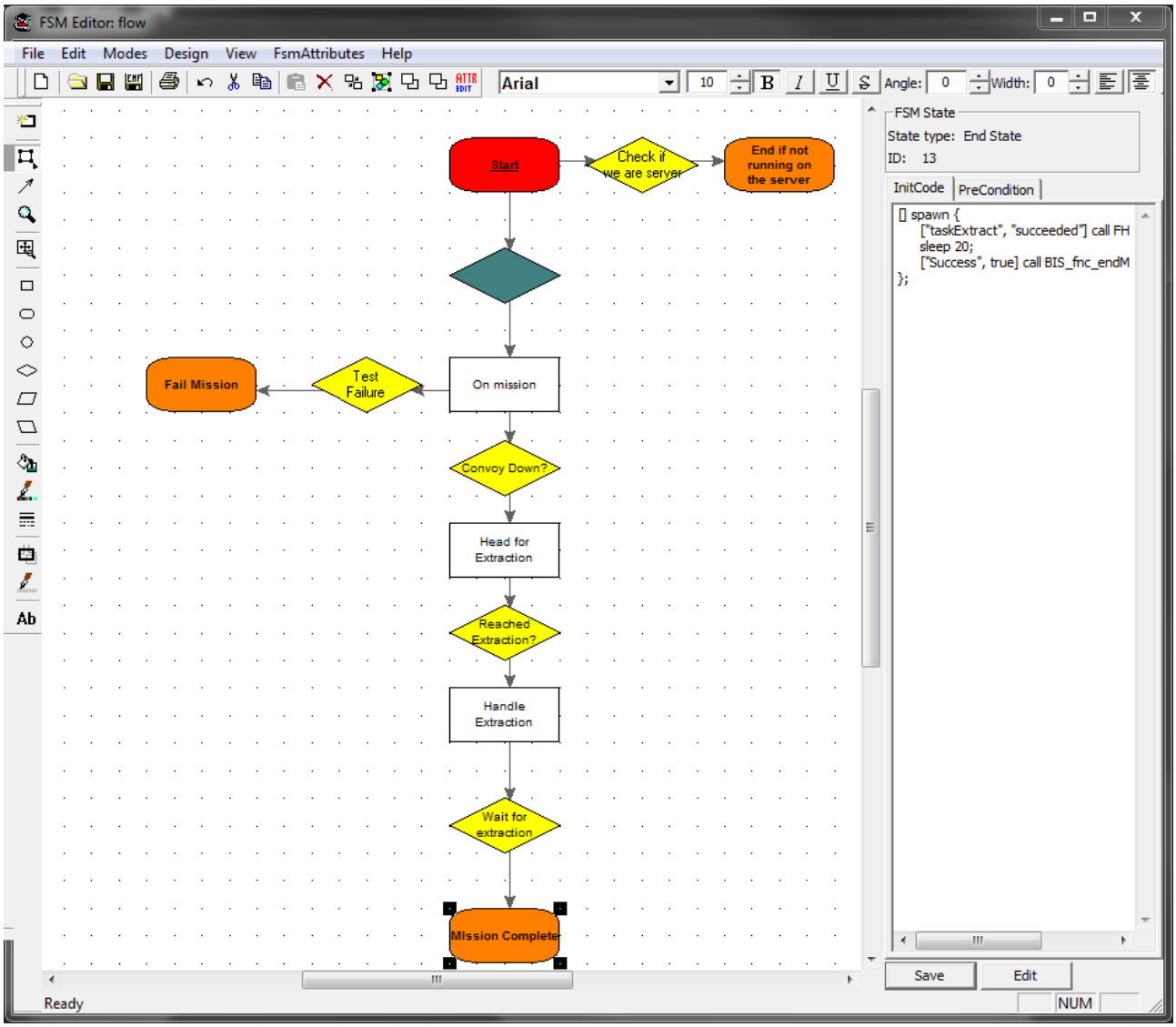
```
!isnil “_extractComplete”
```

This means that as soon as the `_extractComplete` variable has a value, we know that the extraction is complete. Now pull out a new state from this condition, right click on it, and make it an end state. Label it “Mission Complete”, and enter the following code in it (which will be very similar to the one used before):

```
[] spawn {
      ["taskExtract", "succeeded"] call FHQ_TT_setTaskState;
      sleep 20;
      [["Success", true], "BIS_fnc_endMission", true] call BIS_fnc_MP;
};
```

Save the FSM, and test the mission.

For reference, here's how the full FSM looks for me:



Note that this FSM is pretty simple. For one thing, there is no loops in it. At each state, there are only exits from the state, and we never get back to one. There are some inherent difficulties with getting back into a state. Most notably, the condition that we used to exist the state might still be true. For example, consider we had a task for each of the vehicles and would have to check for each individually. Our “On Mission” state would now have four instead of two exits:

1. One to test for failure
2. One to test for the destruction of the first vehicle. The condition would lead to another user state that sets the task to completed, and then leads back to the “On Mission” state via a True condition
3. One to test for the destruction of the second vehicle, with the same basic setup as the one under 2.
4. And finally, an exit that would go out of “On Mission” state to indicate we want to go for the extraction.

There are a number of issues that need to be addressed:

- `!canMove eastVehicle1` will remain true once we come back to the “On Mission” state. That means it will immediately trigger again, and even worse, if this is the first condition, it will never even check the others since it always finds a true condition once the first vehicle is destroyed
- Once both vehicles are destroyed, we will want to have the FSM go through the appropriate branch for that vehicle first before existing the “On Mission” node for the last time. For example, if we destroyed vehicle 1, and got its “Task Completed”, then destroy vehicle 2, we want its “Task Completed” before we go out of the loop with the fourth condition.

To solve the first dilemma, I usually associate a variable with a specific task. For example, I would rewrite the condition of the first vehicle test to

```
isnil “_taskVehicle1” AND !canMove eastVehicle1
```

and, in the state that sets the task to completed, I would add a line that assigns a value (in fact, any value) to `_taskVehicle1`. This way, the next time we reach the “On Mission” state, the first part of the condition is `false` (the variable is no longer `nil`) so that branch is essentially “dead”. The same applies to the second vehicle path.

This also immediately opens up an easy solution for the second problem pointed out above. Now that we associate variables with the specific task, it is very easy to formulate a bulletproof exit from the “On Misson” state:

```
!isnil “_taskVehicle1” AND !isnil “_taskVehicle2”
```

This will be true if and only if we have gone once through both of the vehicle branches. If you want an exercise, try to implement this.

More things that you can do to improve the mission:

- Place a few enemy patrols on the map. You can use the BIS_fnc_taskPatrol to generate some

random patrol route for your units.

- Place sentries on some strategic locations.
- Advanced: Augment the FSM to call in reinforcements if either of the vehicles is destroyed. Place a motorized infantry at Rogain and make them wait at a waypoint near Rogain, with a trigger synchronized to the waypoint using a global variable or whatever other method you want to use.

The basic version of this mission can be downloaded from our web site:

http://friedenhq.org/Downloads/co04_ambush_example.Stratis.7z

Conclusion

I hope that this guide produced some useful information. The most important aspects of successful multiplayer mission making is to be aware of locality and what effects they have on the statements you are executing. The most common issues when running on a dedicated server is that there is no player object on the server, and any script running that references `player` might or might not have an effect, whether desired or not. This guide should have shown how to avoid these problems, and how to achieve a consistent execution based on global or local events. The FSM as a central entity for controlling the mission flow is a useful method to ensure single execution even if an event is triggered multiple times, but again, it must be observed how execution is handled. For example, just calling `BIS_fnc_endMission` in the FSM is not enough; it will then ONLY run on the server. If you are just testing the mission by yourself on a hosted server, that will work because the server and you are the same entity and therefore there is no such thing as a disjoint context, but as soon as you run this on a dedicated server, even if you are testing alone, the problem will become immediately apparent.

It is highly recommended that you set up a dedicated server for testing. I have a separate machine running Linux that I use as a gateway for my home network which I have set up a dedicated server on for testing; to the best of my knoweldge, it is possible to run a dedicated server and a client on the same machine at the same time. Famously, testing only proves the presence of bugs, not their absence, but it is a good (and so far, pretty much the only method) to ensure your mission runs on a dedicated server.

The methods presented herein work on single player as well. You can run the test mission we created in single player as well as multiplayer.

If you have any feedback, please don't hesitate to contact me.

Copyright, Legal Notice and Disclaimer

Much of this document is based on personal experience and collected evidence from the Bohemia Interactive Forum and the Bohemia Interactive Community Wiki. Although the author has made any reasonable attempt to achieve accuracy of the content of this document, he will assume no responsibility for commissions or errors. Usage of this information is at your own risk. The author makes no representation of completeness or fitness for any particular purpose and shall in no event be liable for any loss of profit or other damage, including but not limited to special, incidental,

consequential or other damages.

All trademarks, service marks, product names, or named features are assumed to be the property of their respective owners and are only used for reference, without any implied endorsement when using any of the terms.

© 2014-2015 by Hans-Jörg Frieden. All Rights Reserved.

Contacting the author: I can be reached via E-Mail as Alwarren@ciahome.net or as Alwarren@friedenhq.org, or on the Bohemia Interactive forum under the username Alwarren.