4/28/2013

[FOCK]ERS

FOCKERS ARMA 3 SCRIPTING GUIDE

*Not – do not copy over the text – type it yourself as the " " are different in the editors.

Introduction

Why have I made this?

Well, I've been messing around with Arma and Arma2 for the last couple of years and although there are guides (See links below) out there to help along the way, a lot of learning was done through reading the various forums and getting sound advice from others. It all started after playing many hours of Domination, and then getting fed up of waiting for new versions to come out.

Being all enthusiastic I decide to make my own game from scratch. Realising that it wasn't a straight forward process, I started to understand why any Domination update took so long…A shit load of work is required. However, I battled on and made a whole game mode (Self Advertising – Arma2+OA http://www.armaholic.com/page.php?id=16835) just so that I could understand what was going on under the hood. Although at the time I was quite pleased with myself, I must stress this took ages, and I don't mean weeks, it took at least 6 months to get a sound, none major buggy, version up and running. Luckily my wife didn't mind ☺ either.

Back to why I made this. This really is just as an update to some of the tutorials out there for people willing to learn and who want to contribute to the community. This is not an advance tutorial, this is just to get help to those who have no computer programming knowledge, and just want to learn to script.

Chapter on is fairly wordy, but I have tried to make them as non-computer jargon as possible. People with scripting knowledge really wouldn't need to read this, but any suggestions for improvements from others are welcomed. Some terminology won't exactly fit either but I have tried to make this as non-complicated as possible.

If this manages to help at least one person then I'll be happy.

Regards

Mikie J [FOCK] – BIS forums Mikie boy

*See the end of this tutorial for how to setup a script error reporting system within the game.

CHAPTER 1 – BASICS

Part 1 – Variables Basics

Local, global, public – these are the three main types of variables, however initially we will only be dealing with local variables.

Take a Look at this little bit of code below. Its non-working code, I've just used it as an example.

```
_aiDude = Dave;
_aiDude  playaction "salute";
_aiDude  doMove  (position Eric);
```

First thoughts? WTF!

In this instance we have made a variable or identifier called "_aiDude ". Now we use underscore "_" in front of our word "_aiDude" so that we make this variable local. Without this underscore we make the variable global. This will all be explained later in more detail. In any case "_aiDude" variable has now got all the values of Dave – or can be thought of for now as Dave. All this just by using a single "= " sign; And because it has "_" at the front of it, it is a local variable.

Got that… OK. So what the feck is a variable.

From wiki… "a **variable** is a storage location and an associated symbolic name (an *identifier*) which contains some known or unknown quantity or information, a value."

Variables are names that are used to identify things. Although for now we will stick with singular objects (named objects from editor), values (1, 2, 3, -4, -10, 2.5, etc) and strings ("m16A", "I hate you"); Variables can also relate to parts of code or certain kinds of structured code.

In really simple terms it's a box I put stuff in and then I name that box after whatever is in it or whatever I want to call it. Then whatever is on its right hand side gets put in the box.

Let's say in terms of Arma I have an AI soldier placed in the editor called Dave (I use Dave a lot – not sure why). I need a way to link Dave in the editor to using him in the script. Yes I could just type the name Dave throughout the script when I want him to do something. However, that would mean if I have an AI soldier called Eric I would have to re-write this whole script and change the relevant parts of the script from Dave to Eric so the script and computer know what to do with Eric.

Now, look back to the script

```
Dave playaction "salute";
Dave doMove (position Eric);
```

Two commands (playaction and doMove are Arma commands). Dave will salute, and then Dave will move to the position of Eric (using Arma command position);

Now if we change the above to include _aiDude, this is what it will look like.

```
_aiDude = Dave;
_aiDude playaction "salute";
```

_aiDude doMove (position Eric);

This means I have put Dave in this box called _aiDude. I could have called the box _iLoveSheep or _ericLovesDave, it doesn't matter what I called it, as long as it is local (for now) with an underscore in front. The naming process is just so I know what's going to be in the box. Seeing as we have just put Dave into _aiDude, _aiDude should now have the properties of Dave. Dave is an AI Soldier in the game, and therefore when I say -

_aiDude playaction "salute"; I telling Dave the AI Soldier will salute.

So let's say we deleted Dave from the editor and only have Eric in the game. Without having to go through the whole script I can just change _aiDude = Dave, to _aiDude = Eric.

So when we get to _aiDude playaction "salute"; Now Eric will Salute as _aiDude has got all the properties of Eric and not Dave.

Most of you must be thinking I have no frigging idea what Eric and Dave are doing. And that's fine. Stop thinking of the game and stuff happening in game for the moment. I'm just breaking down how we make a fictitious name and place another object, string, or integer into it. This was my first issue to get over, hence the stressing of the point.

Moving on. Look at the following and see how _aiDude can have anything assigned to it.

_aiDude = 1; integer
_aiDude = "Dave"; string
_aiDude = Dave; object
_aiDude = [Dave,Eric]; array containing Dave and Eric game objects
_aiDude = True; bool – true or false;

Adding numeric value to a variable.

_aiDude = 1;

If this was a script then _aiDude would be given the integer value of one (1).

So I could then do this in the script…

_sum = 2 + _aiDude;

If I was then to print on the screen in game what the value of _sum was, it would say 3 (three). This is because _aiDude has at this moment the value of 1 (one). Thus…

_sum = 2 + 1;

Adding a string value to a variable

If we jump to the next one –

```
_aiDude = “Dave”;
```

This is not the object Dave. The quotes either side of the name indicate in these circumstances that we are referring to a string and not the object called Dave.

If I had this in a script

```
_aiDude   = “Dave”;
_aiDude   playaction “salute”;
```

I would get an error. The game engine would have a hissy fit and will tell me that it expected to make an object salute, but instead it has detected a string with the word Dave inside of it. This is just a word so nothing happens.

Going back to `_aiDude = “Dave”;`

If I printed `_aiDude` to the in game screen, it would just say Dave. No quotes, just Dave. Again this is not referring to Dave game object. I’ve basically asked the computer to print on the screen, in game, the name Dave. It has no meaning and NO link to the game object Dave. I just wanted the game engine to print me the word Dave. I could have easily assigned

```
_aiDude = “print shit on the screen please!***//112230112”;
```

Then when I ask the computer to print this in game on my screen all I would see is

```
print shit on the screen please!***//112230112
```

Again no quotes around it, just the words/letters/numbers inside the quotation marks.

Adding an Array of values to a variable

I will briefly explain this, but I will go into more depth later on as this subject needs a little more input.

```
_aiDude = [Dave,Eric];  array containing Dave and Eric game objects
```

Here, the variable `_aiDude` has the properties of both the game objects Dave and Eric as their names appear and they are not in quotations. It could easily hold a selection of numbers, say [1,2,3,4,5] or strings [“marker1”,”marker2”,”marker3”].

That’s enough for now. Just so you know what they are. As a general concept for now, arrays are those things within two square brackets.

Adding a Bool values to a variable

_aiDude = True; bool – true or false;

Now the value _aiDude has the value of True. It can't have false whilst it's got true. Thus it is mutually exclusive - basically One or the other. True or false are also defined as 0 (false) or 1 (true);

An example of the use of bools is in a while loop function. I'll explain these later, but see how you get on with the concept.
As an example, think of the following daily life events

I wake up,
clean my teeth,
take a dump,
get showered,
get changed,
leave the house,
drive to work,
do very little at work,
return home.

Technically that will keep happening until I get fired (which is possible at this rate) or I die! (again with my job – that's possible).

So let's say I have this bool variable _haveAJob.

Assign it true - _haveAJob = TRUE;

If you used that in a loop it would look like this…

While I have a job, do the following

I wake up,
clean my teeth,
take a dump,
get showered,
get changed,
leave the house,
drive to work,
do stuff at work,
return home,

On getting home, check if I have a job at the end of the day – if Yes it means _haveAJob still is equal to TRUE, therefore - loop back to the start and go through it all again. (Ah crap!!!!!).
However, in the event that I check on returning home, and I don't have a job, we must end the loop as _haveAJob is no longer TRUE, it becomes FALSE. You have broken the pattern, and I would not re-enter the loop, as the principle of the loop running was… while I have a job do the following.

Basically – if I lose my job I would set –

_haveAJob = TRUE; to _haveAJob = FALSE;

That's kind of the thing they can be used for. There are other checks, but again I will explain this in more depth along the way.

Declaring and Naming Variables

If you look up variables/identifiers in other languages – they will generally ask you to assign the type of variable before you can use them. Eg. int number = 0; or string aiDude = "Dave";

In Arma you don't. But personally as a rule once you make a variable a certain type, during its used in a script - keep it the same.

So…

GOOD
_aiDude = Dave; (game object) Then change to
_aiDude = Eric; (game object)

BAD
_aiDude = Dave; (game object) Then change to
_aiDude = "Eric"; (string)

It's always useful to place the variables in a useful location so others can see what they are. Ideally place them at the start. This is not always possible as you create variables as you go along, and sometimes you will use global variables from other scripts. In any case – make them visible to the reader of your script so when looking through your scripts they know what they relate to.

As for naming – local variables must begin with "_". For example _yo or _myVariable. The naming used should not conflict with any command used by the BIS program code (check the wiki here - http://community.bistudio.com/wiki/Category:Scripting_Topics). So you could not use player = Dave; as player is a command used by the game.

As a sound rule, keep to the same naming principles. Start the _local variable with "_" and always use a lowercase letter to start the first word. When you have completed your first word in the variable naming process, the second word, if any required, should start with a capital letter. E.g _myAwesomeVariable. See how the m is lower, but the A and the V are in capitals. This doesn't affect the game but it helps when reading it back.
As a strict rule you cannot use a number to start the naming process, or at least you shouldn't. And you cannot have spaces. If you need spaces use "_". Therefore it should look like this…
_my_Awesome_Variable.

Just for now, add " ; " after any line we add. This is to indicate the end of the line of text that the computer needs to read. As we go on I will explain when to use " ; " and when not to.

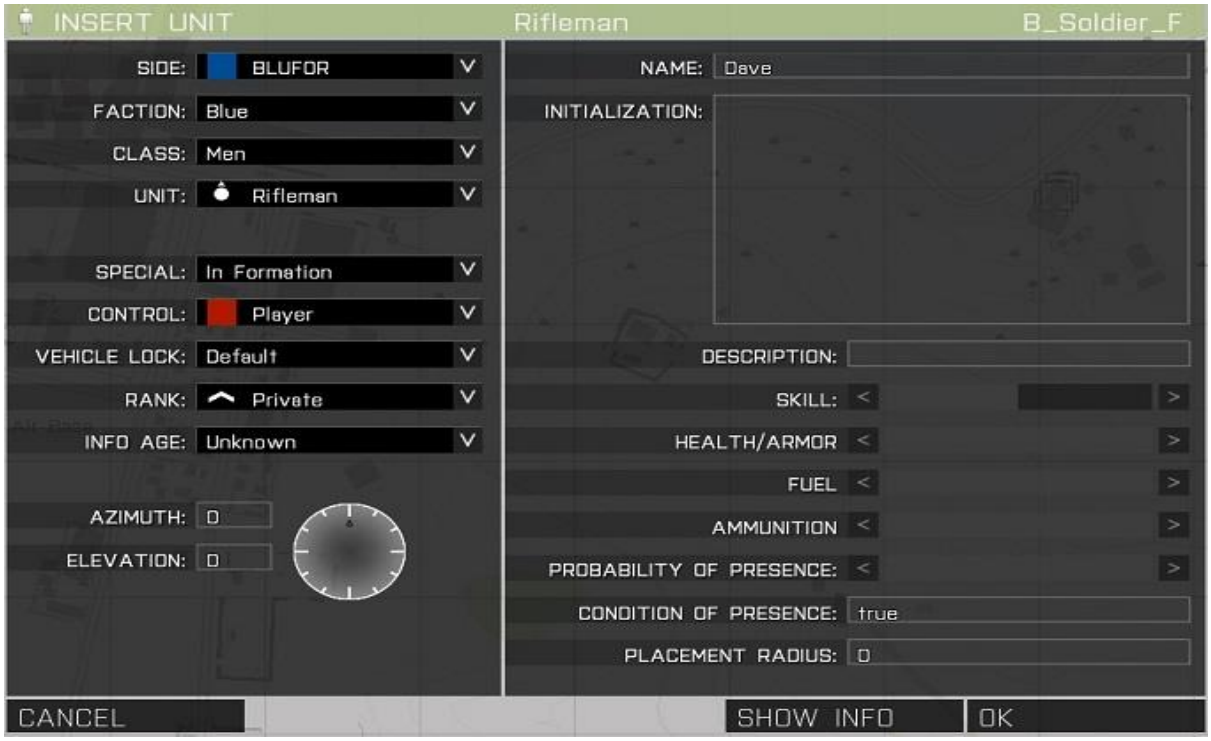
Onto Arma and Variables.

Yes that's right through all this waffling I will now attempt to explain what the feck all that was just about, using the game engine and editor.

*Note – using MP Editor so later on we can add MP scripting.

So Load up Arma 3/or 2, go into play, then into Multiplayer section of the game, then click new, then click ok in the next section, click ok, then select the island (in this case A3 choose Stratis) and then on the <<New – Editor>> on the right hand side (top). Click play and then you're into the editor… (I hope!).

Now you are in the editor. Double Click anywhere on the screen ensuring that [F1] Units has been selected (can do this by pressing your F1 key). This creates an Object player – set his control to player, and yes Name him Dave – with the capital 'D'



Press ok, and then Save the mission as FockTest1 – save as User mission. Preview the game by pressing preview. And your guy will be in the game as you controlling him. Basically player = you controlling him. Press escape, click on suspend, click back until you get back to the choosing missions screen. There you will see your mission on the right FockTest1.Stratis.

Click on that and click edit – that's it you are back into the Editor.
Alt + Tab out of the game and go to the following (or something similar) …

Mydocuments >> then click on Arma 3 or Arma 3 Alpha folder >> then inside that should be a folder call MpMissions. Open that >> inside should be a folder called FockTest1.Stratis.

Double click on that and then inside that folder you will see a file called mission.SQM. This is the folder that is basically a text version of the visual editor. You can edit that mission folder and change co-ordinates etc, and it will affect the game. Anyway – moving on swiftly…

Inside this folder called FockTest1.Stratis - create a new file called hint.sqf – you do this by creating a new text file – opening that text file then saving it as hint.sqf

I use a program called ArmaEdit (http://www.armaholic.com/page.php?id=1455 to do this. Others use Squint (http://www.armaholic.com/page.php?id=11817). Google is your friend. Making a text file and saving it as a " sqf " file does work. Literally, I just tried it, so no arguing! ☺
If using Arma Edit – make sure you click new – then choose sqf – NOT SQS!!!!!

In this file we make a new local variable called _me , then we do…

```
_me = player;
```

This means I have made the variable _me the same as the in game Arma function player. Player function in this case means Person controlled by player (only in single player or when carrying out client side interaction). So now when I use the variable _me we are talking about the person controlling the player.
We then make a hint to our screen – hint is a way of calling texts to the player's screen. I will explain Hints and format later – but for now type…

```
hint format ["player controlled is called: %1",_me];
```

Save this (make sure it is saved as hint.sqf). Now create another file inside the FockTest1.Stratis folder - called int.sqf.

Init.sqf is the file that is automatically executed when the game mission you have made runs. Make sure this is in the main directory of your mission and not any subdirectory of FockTest1.Stratis

In this files type the following…

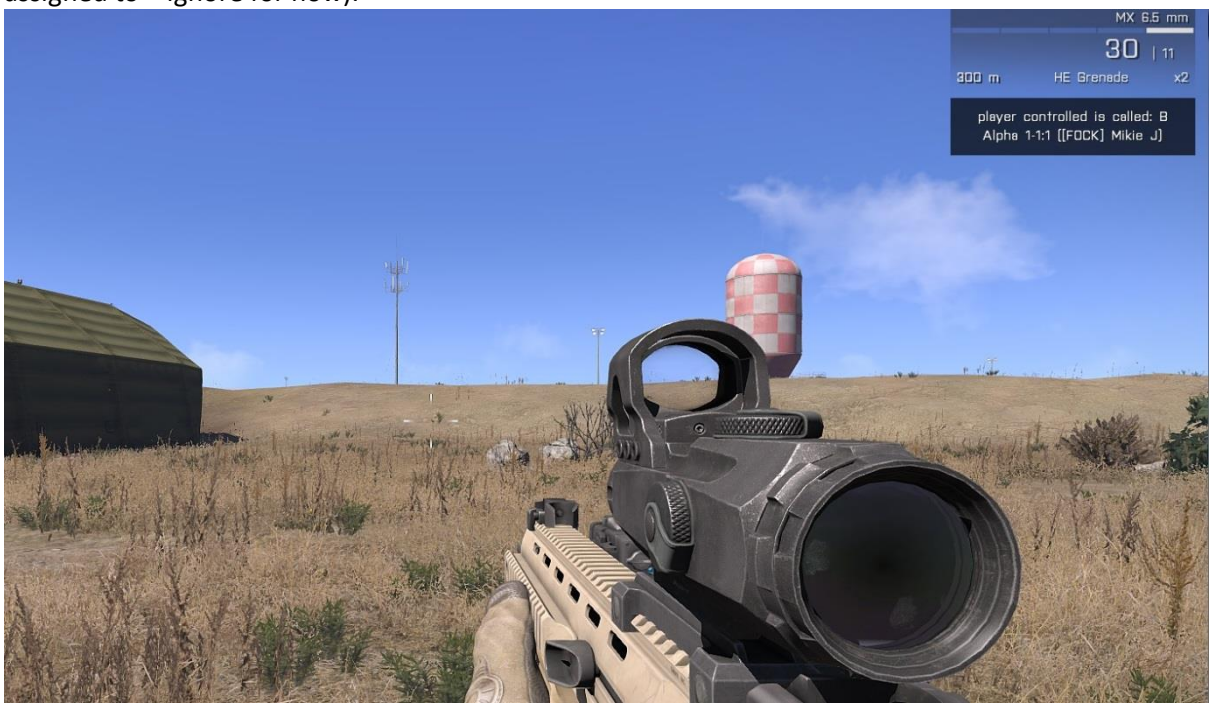
```
execVm "hint.sqf";
```

Save and that's it. Alt-Tab back into the game – should still be in editor screen. You can now press preview and see the hint. If it doesn't show then you have an error. !!

You see how the name that we gave the unit was hinted on the screen. We didn't put Dave anywhere in any of our scripts. Because we named our player Dave in the editor, and used the player command to be hinted. The player command returned Dave as our name. If this confuses you perhaps this will help.

Go into the editor. Delete the Dave name from the player's unit. Now there should not be a name in the name box, it should be empty.
Save and preview again. Do NOT touch any of the scripts.

Now you should have your own name in the corner – plus Alpha 1-1:1 (this is just the group you are assigned to – ignore for now).



Hopefully you can see that the "_me " variable called the player information (in single player – there is only one player – so it knows where to look) - and the player details returned from you were your profile name. It's just the name of the player that has changed and we did not have to do anything with the script. Regardless what you call the player in the editor, the script will check what the variable _me refers to and hints it. (DON'T WORRY IF YOU HAVENT UNDERSTOOD IT – ILL TRY AND ADD TO THE CONCEPT BELOW).

Now let's say you just wanted a greeting screen to come up – and forget about the player name etc.

Let's say on joining you want a welcome hint to appear. We now know that init.sqf runs automatically for the player when the mission runs. And in that init.sqf we asked the computer to execute the file called hint.sqf

So now every time we restart the game that hint.sqf file will be called.

Delete all the contents of hint.sqf add the following…

```
_aHintMessage = "Welcome you numpties to this crappy tutorial";

Hint _aHintMessage;
```

Save, Alt-Tab back to the editor. If you come out of the mission at any time, and then go back to the lobby, ensure you restart each time –When you save the game files/sqf the computer needs to know. Otherwise it loads in the saved version that was saved on exiting the mission the last time you played.

Any way – preview /restart and you should have the message displayed.

Awesome!!!! Lol… It's a start at least.

So looking back at the above, we created a variable called…

_aHintMessage.

We then assigned it or gave it the String information of…

"Welcome you numpties to this crappy tutorial";

So now we know _aHintMessage is a string variable and basically stands for Welcome you numpties to this crappy tutorial, without the quotes.

We then used the in game Arma command Hint to call _aHintMessage.

Hint _aHintMessage;

And as _aHintMessage really stands for the string message Welcome you numpties to this crappy tutorial, we got it displayed on the screen. That is passing a variable to the hint command.
So now we could type…

Hint _aHintMessage;
Hint _aHintMessage;
Hint _aHintMessage;

This would hint the message three times. And look we didn't have to write the whole thing out each time. The _aHintMessage stored the whole string message. If you are actually doing this as a test – it won't show three times as it's so quick – it will look like one constant message.

If we did this in the script…

```
Hint _aHintMessage;
Hint _aHintMessage;
Hint _aHintMessage;
 _aHintMessage = "screw you crappy tutorial";
Hint _aHintMessage;
Hint _aHintMessage;
```

On screen technically we would get …

```
Welcome you numpties to this crappy tutorial
Welcome you numpties to this crappy tutorial
Welcome you numpties to this crappy tutorial
screw you crappy tutorial
screw you crappy tutorial
screw you crappy tutorial
```

This is because we changed the value of what _aHintMessage contained. If you have the same variable and change it to something else, then as you can see above it becomes/represents the last thing you placed in it.

Part 2 – Variables and Basic script editor communication

Now you should be able to hint anything on your screen when you join. Yep, this really is the tip of the slowly melting iceberg.

Now we are going to delete the execVM "hint.sqf"; line from the init.sqf. We no longer want to call the hint.sqf file when to missions starts.

It should now be blank. Save the blank init.sqf and go into the editor. Double click on your player and add the following to his Initialization (or init) box. This is the same as the init.sqf but solely for your player. Whoever controls this player, the init box executes whatever is inside it for him/her when the game starts (more or less). Righty! Add this line to the initialization box of your player…

```
Fock = [] execVM "hint.sqf";
```

Preview the mission and you should get the same hint on your screen. Thus, what happened was the mission was started. It looked in the init.sqf for any stuff to do or execute. It didn't find anything. The mission loaded in and the player's initialization box executed the file hint.sqf. To be completely honest I'm not sure if the init box runs at exactly the same time as the init.sqf. Anyway, after getting the same hint – you can see that the player interacted with the hint.sqf script.

Just to explain that code a little – FOCK represented a handle. The handle is used to identify scripts in operations called using Spawn command or execVM command. At this point just realise you need a

handle to execute a script when called from within the editor/object's initialization screen. In the editor we cannot use local variables, so don't bother. [] execVM is the request to compile and execute a script. The [] represents an empty argument or an array, you can place variables or objects in this which can be passed to the script you are calling – again this will be explained later on. "hint.sqf"; - well this is the script we want to execute enclosed within " ". The file name must be enclosed within the " ", otherwise it will note work.
Rock! So How about some interaction between what we do in the editor and what we do with a script, and how we can start using variables to carry out actions.

Addaction - http://community.bistudio.com/wiki/addAction
I'm sure some of you have already played the game and used an add action. It's the scroll mouse button list of things that appear on your screen. For example when you walk up to an ammo box you get the option to click on gear, or scroll your mouse button to get that option. This is an addaction command. The Arma game has kindly provided this for us and we can use this to link from the game world to a script.

Go back into the editor and delete all the data out of the initialization box for the player. Now add an addaction to the player by typing this into the player's init box.

this addaction ["Call our hint script","hint.sqf"];

Click ok and close the player's details screen down. Click preview. On joining you should not receive a hint when you first join. (If you don't, nice job, if you do - FFS!!! ☺ - you will get errors as you go along don't worry, just go back a few steps and check). Scroll your middle mouse button down (if you haven't got a middle mouse button –Stop! dear lord stop now! Go out and a buy new mouse that has one!) and you should get a box in the left hand side of the screen showing "Call our hint script", and possibly below that an option to change your weapon.

Select the Addaction we just created and you should now have the same welcome hint on your screen. This demonstrates some control we have to call scripts.

Let's Advance a little…

Go back to the editor and then Alt-Tab back to the hint.sqf file. Delete all code so you should have an empty file.

If you read the Addaction link I placed on the tutorial (I know none of you did) you would see that it talks about an array of parameters that are passed. This is where utilising variables comes into play.

In very non-technical terms it can be thought of as follows...

When you use addaction, we know it calls a script. To call a script you have to access it. In relation to the hint.sqf, think of the addaction opening up the hint script and introducing itself, it then says "hey, I'm the player can you hint please." And the hint command passes back to the player and displays on his screen. If you are wondering how it knew to contact the hint.sqf, well we placed that inside the addaction command brackets. Here "this" within the players initialization box just refers to that object its attached to (the player).

```
this addaction ["Call our hint script","hint.sqf"];
```

Here I should mention that the first string ("Call our hint script") is the name that shows up on the screen when you scroll your mouse button to get the addaction list. So change that to reflect what you are going to do. The second string ("hint.sqf") is the name of the script to be called enclosed in "".

Right now for the technical bit. What you didn't see were the parameters passed from the player to the hint.sqf script. The addaction sends three (3) parameters (and some arguments but forget them for the moment) to the script. The parameters are as follows.

Target, caller, ID.

- target: Object - the object which the action is assigned to/or attached to – in this case the player
- caller: Object - the unit that activated the action – you scrolling your mouse button
- ID: Integer - ID of the activated action – the position in the scroll menu – in this case first option.

As the addaction is attached to the player – target and caller are technically the same. The ID is the number where the addaction is in the players list of addactions. To call our hint script - may be located at position ID 1, and to change to a pistol may be position ID 2.

The hint works, as it was a generic hint to the screen for whoever called it. No further interaction was required.

But let's say we wanted to hint who was calling the script. At last a Variable can be used, you didn't read all that crap for nothing.

The addaction technically sent the following to the hint.sqf script…

```
_this = [Mikie J,Mikie J, 1];
```

It declared its own MAGIC variable of " _this " and gives it three parameters to give to the hint.sqf script. It never shows in the script as that, as we can't see it. It just as happens that I know what my player name is, and that I am also the caller.

"_this " is a magic word used by BIS coding structure, and IN THIS SPECIFIC CASE it is an array of things. Again not going into too much depth but in this particular usage it's a variable that holds more than one value. Just remember when you want to choose the first Mikie J/or target – it is in position 0, not position1. Yes there are three items in the array, but they are in the following sequence. [0,1,2]

We need to create three variables to assign each of those parameters to.

Type the following in the hint.sqf

```
_target = _this select 0;
_caller = _this select 1;
_id = _this select 2;
```

The variable _target has been given the first passed parameter from the addaction - the value Mikie J.

POS = 0 , 1 , 2
_this = [Mikie J,Mikie J, 1];

What you technically have done is accessed "_this" – which is an array or list containing (Mikie J , Mikie J, and 1) , and you have asked it to select position 0 from that list. Well in position 0 of this particular array is the first Mikie J. So looking at "_caller" variable – you have asked it to select position 1 from the array which contains (Mikie J , Mikie J, and 1) and selected the second Mikie J. And then for _id you have passed the third parameter number 1 to it, by selecting position 2.

Ok, so now we should have

```
_target = _this select 0;
_caller = _this select 1;
_id = _this select 2;
```

Add the following..

```
Hint format ["Target = %1",_target];
```

Save the hint.sqf, Alt-Tab back to the game and preview or restart the mission.

Now when you load in scroll to the addaction, call our hint script, and select it. You should have a hint on the screen. Which says something like…

Target = A1:1-1 [FOCK]Mikie J

Go back to the editor, Alt-Tab into the hint.sqf file.
Now change the hint line in the hint.sqf to the following…

```
Hint format ["Caller = %1",_caller];
```

Save the file and Alt-Tab back into the editor. Preview or restart the mission.

Now when you load in scroll to the addaction once again, call our hint script, and select it. You should have a hint on the screen. Which says something like…

Caller = A1:1-1 [FOCK]Mikie J

Looking back on it all, what have we done?

Breaking it down – we have made the addaction in the player's init box. We scrolled the mouse and selected the action to call our script. On selecting the script, the addaction from the player has passed an array containing [the player who is the target or to whom the addaction is attached to – [FOCK] Mikie J, the player who is the caller (who physically called the script)– [FOCK] Mikie J, and the addaction id position 1].

That array of information has been assigned the variable - "_this";

_this = [Mikie J,Mikie J, 1];

* I've removed the [Fock] bit from my name to save typing that's all!!!

We have then created three variables so that we can assign them the parameters from our "_this" array.

_target = _this select 0; This is the first thing in the list of _this – which in my player's case is me - [FOCK] Mikie J.
_caller = _this select 1; This is the second thing in the list of _this – which in my player's case is me - [FOCK] Mikie J.
_id = _this select 2; This is the id number – which is the position on the scroll down menu for the addaction in game.

Now that our variables have been filled with the above values, we then pass them to our hint command.

```
Hint format [“target = %1”,_target];
```

We could also do the below for hinting the _target as it would produce the same results – After all _target = _this select 0;

```
Hint format [“target = %1”, _this select 0];
```

They are both pointing to the same thing. The variable we defined just makes it easier to know what we are calling.

*HINT FORMATTING - Notice how _target is the last thing in these brackets. Here it is being a referenced so that it can be placed inside the “ “during the in game hint, the position of where %1 should show up on the hint screen.
You notice that ‘%1’ is never in any of the hints within the game – that’s because the variable _target is being called in its place.

The script is then called and a hint of the above is sent back to the player to hint on his screen. Depending on the person who is the target or caller – depends on what is returned in the hint.

*TARGET AND CALLER - Just a quick explanation of target and caller – if you placed the addaction in an ammobox init box instead of the player’s init box. The addaction in Single player would appear on the screen when you walked up to the ammobox and scrolled you mouse button. This is because it’s attached to the ammobox now and not you. So if you click on the call hint script – you are the caller and the target is no longer you, it’s the ammo box.

I will explain further me thinks.
Do not touch the script – instead Alt-Tab back to the editor. Double click on your player and delete the addaction line. Create an empty object, class ammo, unit - any type of ammo box. Inside its init box, type the following…

```
this addaction ["Call our hint script","hint.sqf"];
```

Click ok and then preview.

On connecting to the game – look away from the ammo box and scroll your mouse button. You shouldn’t have any addaction to do with our hint.sqf script. You may have a weapon change one or something similar, but shouldn’t have the “call our hint script” addaction. If you do check you haven’t left the code in the player’s init box. If you don’t - nice one. Now walk up to the ammo box, scroll you mouse button and you should have an addaction to call our script. Select it and see what you get…

Since the script Hint.sqf should have been left with “ _caller “ being passed to the hint function, it should show your name as caller in the hint box. This is the same as the last time we tried it. But remember we haven’t got an addaction attached to us – this is all from the ammobox and its addaction. The ammobox has detected who the caller is and placed that in its _this array as the caller parameter. Then it’s sent the following parameters to the hint.sqf.

```
_this = [ammoboxmodel, [FOCK] Mikie J, 1];
```

Now exit back to the editor, Alt-Tab to the hint.sqf script and change the hint format line back to show…

```
Hint format ["target = %1",_target];
```

Save the hint.sqf. Alt-Tab back into the editor and then preview/restart the game. Do the same again whilst looking away from the ammo box and see if you have an addaction. Hopefully not. If you don't have an addaction check the player's init box(AGAIN!!! Lol) and make sure you are not near or looking at the ammobox in game. Right, now we walk up to the ammo box again and scroll the mouse button. What do you think it will show in the hint box for: Target = ????

Target = 3f0e6b00# blah blah

Yep – its showing the variable defined as "_target" - the target to which the addaction is attached to – in this case the Ammo box. It is no longer attached to us, so it shouldn't say we are the target. Yes, we are still the person calling it, but not the target. As for that Target = load of numbers – this is just the model name and details for that particular ammo box model. You may have something slightly different. If I gave this mission to a friend called John and he played it, when he walks up to the ammo box and uses the addaction the target hint would stay the same – as it's still the same ammobox, but the caller would be John, because he is now the person activating the script.

You can place these addactions anywhere. You can place the addaction back onto yourself. And this is the clever bit…

Remember `_aiDude = dave;`

I said if we used the above, we would have to make a script specifically for him, and then when it came to Eric we would have to create another specific one for him. HOWEVER, now that we have an array of parameters passing things to our script, namely the target, the caller and the ID – For each object that the addaction is attached to it would place different parameters in its array, thus producing different results.

Notice how we didn't change the hint.sqf script when moving the code from the player's init box to the ammobox init box. We did not have to change anything so that _target =_this select 0; would work. It stayed the same. This is because the script can be called by anyone. _this select 0 just depends on who is sending stuff to the script. Just so that I'm not confusing anyone, think of it this way in the following example if it was done from scratch...

I've made only one hint.sqf (above) file. Ive placed one addaction in my player's init box and another addaction is placed in my friend's Paul player init box.

Basically I copied this addaction ["Call our hint script","hint.sqf"]; into both of Paul's and my init boxes in the game editor.

We both join the game and we both have an addaction saying to call our hint script (remember I only have one file called hint.sqf). If we both independently select the addaction, we both independently send our different "_this " arrays to the hint script.

For me it would be _this = [mikie j, mikiej, 1];

And therefore _target = mikie j;
_caller = mikie j;
_id = 1;

And for paul his _this = [paul, paul,1];

And therefore _target = paul ;
_caller = paul ;
_id = 1;

We can both simultaneously access the same script and produce different results. By me using the addaction it has nothing to do with paul accessing the addction. By me using that addaction I do not change what _target is defined as when Paul calls the script. I only affect myself.

Further explaination – and trying to hit this point home….

If we had the _caller hint selected in the hint.sqf –

My hint box on my screen would say…

Caller = [Fock] mikie j

Paul's hint on screen would say…

Caller = paul

Thus, I didn't have to make two scripts, I just used the one, and passed the parameters from the addaction into the script – just depends on who was calling and who was the target.

I'm hoping from all this you can now understand why we use variables and what their purpose is for. This is not the most in-depth way of telling you, but I thought this would be the best way to do it.

Part 3 – Variables and Basic scripting commands

Read this when you can: http://community.bistudio.com/wiki/Control_Structures

Let's start with something easy. " **IF** "…
'If statements/conditions' can be thought of as they sound. If shit happens then do stuff!
We break it down to - if a condition is met we do the following stuff. If that condition is not met, skip over it and move on. The condition has to be met and the test for the condition has to be a Boolean

TRUE or FALSE.

For example…

if (3 is greater than 2) - TRUE
if (3 is less than 2) – FALSE
if (3 is equal to 2) – FALSE

So you can see we need a true or false to be returned from our condition - BOOL variables. Let's make a new script file called ifStatement.sqf

Inside that file let's make a Bool variable called _isPlayerAlive, and make it true.

```
_isPlayerAlive = true;
```

Then we type…

```
If(_isplayerAlive) then {hint "alive";};
```

Now, go to your blank init.sqf and add the following line, so that our ifStatement.Sqf gets executed when the missions starts up.

```
execVM "ifStatement.sqf";
```

Save the init.sqf, Alt-Tab back to the mission editor and press preview.

When you join you should have…

Now, at present _isPlayerAlive has nothing to do with our player. It's just a bool variable that we made up and set it to true. We could have called it _missPiggy = true.

If you ran the script again changing _isPlayerAlive with _missPiggy so that it equals TRUE and then inserted _missPiggy into the 'if condition' instead of the _isPlayerAlive variable …

```
If(_missPiggy) then {hint "alive";};
```

You would get the same result. A hint saying "Alive". This is because you have technically written - if _misspiggy is equal to true then hint "alive"; Well as we have set _misspiggy to true it is therefore correct.

Moving on… Escape back to the editor then Alt-Tab back to your ifStatement.sqf.
Delete all information from your file, so it's now a blank script. Type the following…

```
If (alive player) then {hint "I'm alive!!! woohooo"};
```

Save the file. Alt-Tab back into editor and preview the mission. You should have the hint shown on your screen. I changed the hint as I wanted to show you that it has changed.

So what happened there? We certainly did not set any variable to true or false for the hint to show. Well in this case we used the BIS command name 'alive' and the command 'player'; it is at this point I invite you to visit http://community.bistudio.com/wiki/Category:Scripting_Topics

This page is invaluable. It is a library of all the BIS defined functions and names used in the game. Save it to favourites now ☺. Type player into the search box on left hand side and press search - look at the page.

So what do they mean together? Player is defined as "Person controlled by player". And it states the Return Value is given as an Object. This means the word 'player' is an object and that the computer, in this case, will return a result of us the player. In multiplayer there are going to be lots of players, and the use of the command player becomes somewhat complicated. However for now, since this is still billy basic single player scripting, player can be thought of as us the controller.

The example given on the web page is player addrating 500;

The placement of 'player' at the start of the code is important in the use of this particular code, as it is dictated to by the command addrating. If you wrote addrating player 500; it wouldn't work as addrating expects an object (in this case a player) to be placed in front of it.

If you type alive into the search box and search it you see that the example here has 'alive' at the start of the code before 'player'. Don't worry about the ?!(..bit) – just a shorten way of saying if not(). In this instance 'alive' requires an object to check. It could have read alive Dave, or alive radiotower, or even alive _ammobox.
Look at the return value; here it stipulates a return of a Boolean (true/false). "True when alive, false when dead".

Use this webpage to assist with alignment and order of words!!!!!!

Back to our script. So what is it actually doing?

If player is alive then hint message.

Alive requires an object to test whether it is alive or not. We gave it player - Player in this case returns us, the human object.
So it can be thought of as…
Return the object attached to player character – which is us. Now check if the object player is alive. The word alive does its own hidden calculates to determine if we are alive or not. If player is alive return a Bool of TRUE, if he is not alive return a Bool of FALSE.

Thus, in the case of our script we know we are alive as we have control over our player. The script asked to check if we are alive, and if so hint a message. We got a message so we must be alive.

Changing the code to

If !(alive player) then {hint "I'm alive!!! woohooo"};

This will not produce any results. Simply by placing '!' in front of the brackets, it is asking to reverse the test inside the brackets to 'NOT'. Thought of as… if player is not alive then hint a message.

Save the script. Alt-Tab back into editor and preview it. You should NOT get any hint. That is because it has tested whether we are alive or not. Returned that we are alive, and therefore the condition of 'if NOT alive player' has NOT been met – so the hint will not be executed on the screen.

As I said before…

If shit happens then do stuff!

Which translates to - if your condition is met then hint on the screen. If it is not met then skip it. Unfortunately I can't show you the reverse with the player, as you won't get a hint if you are dead at this stage of scripting. But if we Alt-Tab back into the editor, open up the properties of the ammo box (double click on it), and in the 'name box' type AmmoSups. AmmoSups is just a random word by the way – used as a naming process for our ammo box. Click ok, then click on your player and open his properties. In his Initializtion box type the following…

this addaction [“ammo alive”, “ifstatement.sqf”];

Click ok, and Alt-Tab back to your ifstatement script.

Now change the code by removing player and replacing it with AmmoSups. Change the hint message also.

If !(alive AmmoSups) then {hint “The ammobox is not alive”};

We are now asking… if the object AmmoSups is not alive then hint message. Save the file, Alt-Tab back to your editor and click preview. At this stage you should not have a hint shown when you join. If you scroll your mouse button to the ‘ammo alive’ addaction and click it. You should not get a hint.

Why? Because the box in front of you is working (you can still access its gear menu) and is alive. Alive in this instance does not mean laying on the floor blood pouring out of you, legs twitching. It is just a test to see whether its defined state (as defined by BIS) is set to Not Alive or Alive.

Now point your weapon at the ammo box and shoot the crap out of it until it starts smoking. Its gives a poor fireworks display and then adds some smoke and then slowly disappears. Now scroll your mouse button to the ‘ammo alive’ addaction again, and see what you get. You should have the message hint up on your screen. Again why? Because in this instance you shot the shit out of the ammo box, in turn destroying it, which in turn makes it NOT Alive. The computer checks to see if the object we named AmmoSups was alive or not and it was not. Therefore condition met and it gave us the necessary hint.

Again…

If shit happens then do stuff!

Hopefully by now you should have a grasp of the ‘if - statement’. If not read on and ill jabber on more about the process. I know, you are living the dream!!! Or not as the case may be.

You can test more than one condition within that ‘if condition’.

Say we had a goal of making sure that if we are alive and the ammobox object (AmmoSups) is alive then hint the message for us. To do this exit back to the editor, Alt-Tab back to our ifstatement.sqf. Delete the line that was there and add…

If ((alive AmmoSups) AND (alive player)) then {hint “The ammobox is alive, and so are we!”};

This is asking for two conditions to be met before it will hint the message. Notice how I have wrapped the two alive conditions into their own brackets, and have then wrapped them both within one big bracket. The computer doesn’t necessarily need these internal separator brackets, but this is to allow easier reading for humans. There are times when you need these brackets especially for arithmetic conditions and order of assignment, but generally for things like this it’s just to make it easier to read. Also note how I’ve use alive twice, placing it next to each object that is required to be tested. You cannot have one alive for all i.e if (alive AmmoSups && player) – this does not work!!!!

Save the file, Alt-Tab back to editor and preview. You should now have a hint saying “The ammo box is alive, and so are we” on the screen when you start. Once again scroll your mouse button to the ‘ammo alive’ addaction and click it. You should have the same hint on the screen. Or it may look like

it hasn't changed. Either way it's confirmed that both the ammo box object named 'AmmoSups' is alive, and the player object (us) is also alive. Therefore condition met – hint the message.

Right, following so far? Two 'TRUE' conditions met. What happens if one of those conditions is not met and therefore is FALSE? Destroy the ammo box again. Wait until the box is alight, then scroll your mouse button to the 'ammo alive' addaction and click it. This time you should not have any hint. Why? The ammo box is NOT alive, but we are Alive. The condition required was two TRUE Booleans. Well we know the ammo box is not alive and that this returns a FALSE Boolean. Thus the condition cannot be met and the hint will not be shown.

Remember we used 'AND' (you can use && as well instead of AND), so if one of the Booleans was false then the condition cannot be met.

If the ammo box object is alive AND the player object is alive then hint a message.

So what happens if we remove the 'AND' and replace it with an 'OR' (can also use || - left of your z key).

Exit the game back to editor, alt-tab back to ifstatement.sqf.

Remove the AND and replace with OR so it reads…

```
If ((alive AmmoSups) OR (alive player)) then {hint "The ammobox is alive, and so are we!"};
```

Save the file and alt-tab to the editor, press preview. Once Again blow the crap out of the ammo box. Wait until the box is alight, then scroll your mouse button to the 'ammo alive' addaction and click it. This time you should see the same hint or it remains the same as when you joined. Why? The ammo box is NOT alive, but we are Alive. The condition required was One TRUE Boolean. Well we know the ammo box is not alive and that this returns a FALSE Boolean. Thus the condition can be met as we are alive. Basically - if either the player or ammo box is alive then hint.

Hopefully, the 'if statement/condition' process should becoming clear to you. It can be used for various tests to which can be helpful when creating a game. For example, say you have a variable called _count. _count holds the amount of friendly units in the game, and say you wanted to spawn

more friendly enemies if that number falls below 3, then the if statement/condition process does that simply.

let's add some excitement (cough, cough) to this tutorial and create that counter.

Exit the game back to the editor, click on your player and access his property screen. In his Initialization box, delete what is there and add another addaction line…

```
this addaction ["add Friendly AI", "ifstatement.sqf"];
```

Yes you could have just changed title of the addaction from 'ammo alive' to 'add friendly AI', but it is good way to improve your knowledge of scripting. So, click ok, moving on…

Alt-Tab back to your ifstatement.sqf and delete what is there so we have an empty file, then add the following…

```
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
```

Save the file and alt-tab back to the editor. Click preview and then scroll your mouse button to the addaction "add Friendly AI". Click on it and see what happens. Yep a soldier is created on your screen. Do this again a few times and you will see a shit load of AI that are spawned.

As you can see, for each time you scroll you mouse button and activate the addaction to create a new soldier, a new soldier is created. This could get messy if you add too many, and if you add way too many your computer will have a fit. Thus, back to our restricting of soldiers script.

Escape back to the editor, Alt-tab back to the ifstatement.sqf.

I will briefly explain how the spawning of a soldier worked…

```
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
```

*Createvehicle should not generally be used to create units that are live or need to be. I have only used these as an example. We will use createunit a bit later on….Use createvehicle command for vehicles and ammo boxes etc…

Jump on the wiki to this page… http://community.bistudio.com/wiki/createVehicle
It states "Create an empty object of given type classname at given position". That's exactly what we did. Breaking it down, we created a variable called _newSoldier. Then we assigned the code

"B_Soldier_GL_F" createVehicle (position player); to it.

The first part "…Create an empty object of given type classname" – this is the String "B_Soldier_GL_F". Here we get the class name (as given by BIS) of a soldier (ARMA3) then we tackle the "…at given position" bit. We use the built in command name of "createVehicle" to initialise the creating of the soldier, and then we give the "createVehicle" command the position it requires. If you look on the wiki it states "position: Array - format Position to create the vehicle at".

Click on http://community.bistudio.com/wiki/position to see what a position means.

Well in short, by using the position player command (and notice where 'player' is located in the alignment – it comes after position) we get three co-ordinates returned x,y,z – which is our player in 3D space within the game.

When you type in 'position player' after the 'create vehicle' you are asking it to create the new soldier at the returned co-ordinates (x,y,z) of your player. You could have changed 'player' for 'AmmoSups'. 'position AmmoSups' would have returned the x,y,z co-ordinates of the ammo box, and therefore the soldier would have been created at its x,y,z co-ordinates. So that's how that command 'createvehicle' and 'position' work.

NOTE - Always look at the wiki for the specific command you are using and see where the object (in this case 'player') is required to be placed – that is before or after the command. Again Createvehicle should not generally be used to create units that are live or need to be. I have only used these as an example we will use createunit a bit later on….

So, back to the restricting of the Spawning of the new AI. Add the following and I'll explain after…

```
if (counter < 3) then
{
 _newSoldier = "B_Soldier_GL_F" createVehicle (position player);
 counter = counter + 1;
};
```

Save the script and open up your init.sqf. Delete the line…

```
execVM "ifStatement.sqf";
```

…as we are accessing this file by way of the addaction (this addaction ["add Friendly AI", "ifstatement.sqf"];), and don't need it to load up on startup anymore.

Type this in the init.sqf…

```
counter = 0;
```

Save the init.sqf and alt-tab back into the editor. Click preview and use the addaction add Friendly AI. You will see that an AI soldier will spawn, try it twice more and two more AI will spawn. Try it a

fourth time and nothing happens. Tada – we have restricted the spawning of AI. Escape back to the editor and Alt-Tab back to the ifstatement.sqf.

Explanation required.

if (counter < 3) then…

This is again is a condition that needs to be met. Well firstly if you can't remember GCSE/High-school maths – what the feck is '<' ? It is a mathematical sign.
'<' means less than.
'>' means greater than.
'<=' means less than or equal to.
'>=' means greater than or equal to.

The condition to be met then is, if counter is less than 3 do….create a soldier. We placed counter at 0 in the init.sqf. By placing it there it starts off at 0 when the game loads up in single player. There is a better way to do this but I will leave that for the multiplayer bit, as this process is quicker.

Anyway, we created a global variable (taken from wiki "…Global variables are visible on the whole computer where they are defined. Names given to units in the Mission Editor are also global variables pointing to those units (for example our ammo box named AmmoSups), which may not be redefined or modified. Identifiers of global variables *must not* start with underscore"). Basically your computer can access that variable whilst in another script. Thus our computer is fully aware on the very first usage of the addaction (to call the ifstatement.sqf), that counter is equal to 0 (zero).

So the 'if statement' really reads…

If (0 < 3) then do…

Simple - 0 is less than 3. Moving on… we know this condition is met and so the script goes onto the next part. It defines a LOCAL variable (notice the underscore – which makes it unavailable to other scripts. It is therefore only local to this script and only this once as the script does not have a loop, and therefore finishes once we have either managed to create an AI or not. In essence when we run this script again the only thing the compute will remember will be the value of the counter variable as it is global).

The local variable _newSoldier assists us with creating a new soldier as already explained. So what happens next? Well below that we have…

counter = counter + 1;

Here we see 'counter' being assigned the variable counter +1. WTF?

Well. counter is already defined as 0, so it should look like this.

counter = 0 +1;

Which obviously totals the value of one (1). It is saying - assign to the variable counter, the current value of counter (so we don't wipe it), plus the integer one.

Say counter equalled 24.

counter = 24.

counter = counter + 1;

Which would look like…

```
counter = 24 + 1;
```

So now counter holds the value of 25. Thus, anytime we refer to 'counter' is will hold the value of 25 until we change it.

If we had just written …

```
Counter = + 1;
```

Then did …

```
Counter = +1;
```

Counter would equal one (1), not two (2) as thought.

All we are doing is wiping whatever is in the variable counter and replacing it with the integer one (1). It would always be one (1). The reason for this is you are defining the one (1) as a positive number and not a negative number one (-1). That's it; no actual math is taking place.

So if you now look at the script is it states the following... 'If counter is less than the value 3, then create a soldier at the position of the player, then increase/increment the variable counter by one (1). Then exit the script as there is nothing else. 'The variable counter is global (let's say the computer stores this variable in its longer term memory outside of any script) and can be changed by this script, unlike any local variables.

If we run the script again it says the exact same. But this time counter has the value of 1. However 1 is less than 3, so carry on as the condition is met, thus create another solider, and 1 is added to the sum of the counter variable.

We run the script once more and we get the same result, but this time counter has the value of 2. As we know 2 is less than 3, so carry on as the condition is met, please create another solider.

We run the script one last time, but this time nothing happens! Why? Well this time counter has the value of 3. Unfortunately 3 is NOT less than 3, so you can't carry on. Alas the condition is NOT met. Since it's not met, the code after the 'if statement/condition' between the '{ }' does not get executed. It is simply passed over and that is the end of the script. No matter how many times you run the script you will never get another AI to spawn, as counter is equal to 3, and the condition to spawn the AI requires that counter be less than 3.

If we changed the script to counter < 4, we can spawn 4 soldiers (remember first one is created when counter = 0), but that's only because we exited the game and changed it. Once the game is running and you have set that condition to be less than 3, you can't change it unless you have used a global variable in place of the 3, and you have a script or code that allows you to change that global variable.

We are not going down that route just yet, but as an example…

```
AIrestriction = 3; // this can be changed by another script if required.

if (counter < AIrestriction) then
{
 _newSoldier = "B_Soldier_GL_F" createVehicle (position player);
 counter = counter + 1;
};
```

I will return to global variables later on in this tutorial.

Right, now we have a working spawning script of soldiers. And we now know that once the condition is not met nothing will happen. Wouldn't it be nice to know when the condition is not met rather than bugger all happening? Well that's when the 'Else' command assists.

ELSE

Remember this…

'If statements/conditions' can be thought of as they sound. If shit happens then do stuff!

Well… 'Else statement/conditions' can be thought of as they sound. If shit happens then do stuff! Else, if not, do some other stuff.

Fairly straight forward. Basically 'Else' can be thought of as mutually exclusive, it cannot do both. For example, it can't be night and day in the same place at the same time. So, transferring that to our newly created script we would add an 'Else' command to the end of our current 'if condition'. All we do is remove the last ';' which follows on from the '{}' brackets. This in turn opens up the script. If we don't add anymore code to this script then we would get an error. Thus here we add the Else component.

If (condition) then {hint "condition met";} else {hint "condition NOT met";}; You will see below how I have set out the brackets (or scope). They are offset to assist with easier human reading; again this is not for the computer's benefit.

In the ifstatement.sqf – amend your script to show the following.

```
if (counter < 3) then
{
 _newSoldier = "B_Soldier_GL_F" createVehicle (position player);
 counter = counter + 1;
} else
      {
      hint "Sorry you have reached your maximum number of AI";
      };
```

Save the script, Alt-Tab back into the editor, preview the mission and call the addaction three (3) times to get three units. Then try it once more and see what it shows on the screen.

Once those AI have reached the number of three (3) – which in turn the variable counter has reached three (3), then no more can spawn. When running through the script, the condition is met three (3) times and the code in the first set of brackets (before Else) is called. On the fourth call to this script, the condition is not met therefore the first set of brackets are ignored and the second set of brackets (after Else) are called, which give us our hint on the screen of "Sorry you have reached your maximum number of AI".

The if/else usage is a might handy tool to use. You can even have a nested if statements like…

```
If (condition met) then
        {  do stuff  } else {
                                If (different condition met) then
                                {
                                  Do other stuff
                                }
                                  else
                                       {
                                        Do some other stuff
                                       };
                          };
```

Notice the red code is inside the first else scope. It then has another If-else condition inside it.
Like I said handy tool, but try not to complicate it all. You could have a load of straight forward if else statements to test certain things if needed be – or you can use them as hints in game whilst testing.

And just to tidy up…you can have if statements nested within if statements or within else statements. *taken from BIS website.

```
if (alive player) then
{
      if (someAmmo player) then
      {
       hint "The player is alive and has ammo!";
      }
       else
           {
            hint "The player is out of ammo!";
           };
}
else
  {
   hint "The player is dead!";
  };
```

Hopefully something is starting to sink in at this point. If not take a breather, place a load of civilians of the map, save the game and load back in and lob grenades at them – always something to cheer you up.

Part 4 – Loops and scripting commands

Ok, so now we are moving onto the looping of information and coding. Remember when I spoke about looping my daily life…

I wake up,
clean my teeth,
take a dump,
get showered,
get changed,
leave the house,
drive to work,
do very little at work,
return home.

And I said that it works by - if a condition is met or not met then it will end. The condition was - if I had a job at the end/start of the day, do another loop of the above. Well that's what we are going to do here. Not my life that is, looping.

So if you are not in the ifStatement.sqf, Alt-tab into that file and delete or comment out all that's there.

We are staying with the recruitment process, and staying with the restriction. Remember in order to create one (1) AI we can use the command line of…

```
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
```

If we wanted to create three (3) AI in one go without having to scroll to our addaction three (3) times then we could just type…

```
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
```

And that would do it.

Obviously if we wanted to do anything more with each one of the AI we would have to change the variable _newSoldier to something slightly different each time so we can reference it properly. For example…

```
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
_newSoldier1 = "B_Soldier_GL_F" createVehicle (position player);
_newSoldier2 = "B_Soldier_GL_F" createVehicle (position player);
```

All I did was added a number to the variables to make them different. So now, in relation to the second soldier we made, in the same script I could refer to him as _newSoldier1. And then say for type…

```
If (alive _newSoldier1) then {blah};
```

This checks if soldier 2 is alive. If we did not change the name of the variable of each soldier, when we type _newSoldier it will always refer to the last soldier made, as the variable has been over written.

All is well for doing this with three (3) soldiers, but say you wanted twenty (20), that's a lot of unnecessary code. Therefore is there a way we can do this without typing shit loads? Funny if I said no at this point… lol.

FOR loops

First type: forspec type.

As per the wiki…

```
for [{_x=1},{_x<=10},{_x=_x+1}] do {};
```

Self-explanatory I think, lol.

From left to right can be thought of as follows

For [{starting point}, {condition to be met – like the if statements}, {step/increment}] do
In English… start at the starting point, and while the condition is met, step the loop up or down.

```
for [{_x=1},{_x<=10},{_x=_x+1}] do {};
```

In this example _x=1. So we are setting up our variable to be used (_x to equal 1) as our starting point to count. Then we are saying while variable _x is less than or equal to 10 (which obtains a Boolean), then we step or add one (1) to the variable _x. We then go on to the 'do' part and carry out that code within the '{}' brackets. We only execute this code once at this point. Once the code in the 'do' brackets is complete, it will then loop around again checking if the condition has been met. It will then do this until _x is greater than or equal to 10.

As a good test add the below to your ifstatement.sqf…

```
for [{_x=1},{_x<=10},{_x=_x+1}] do
{
_newSoldier = "B_Soldier_GL_F" createVehicle (position player);
};
```

Looking at this script this is what happens. We create the variable _x and assign it 1. Then we test if _x is less than or equal to 10, well yes it is. So we then add +1 to _x, so now it equals 2. We then jump into the code part where we create a soldier. We assign the variable _newSoldier to the createVehicle part, which creates the soldier. Once the creation of the soldier is complete we don't assign anything to _x because the loop has started – so once loop started you can think of {_x=1} as not existing.
At this point _x is equal to 2, which we know is less than 10, condition is met, then we step _x up one by adding another integer of 1 to it. So now _x equals 3. We then jump into the code bit of the 'do' brackets and create another soldier - _newSoldier variable is overwritten. Once this soldier is created, go back to the start of the so called 'for loop', check the condition, if condition is met, which it is as 3 is less than 10, we then add another integer value to _x, so now it equals 4. As the condition was met we jump again into the creating of the AI soldier, and loop back around again.

This goes on until _x = 11. To see how this works, with the above code saved in the ifstatement.sqf, Alt-Tab back into the editor – click preview, and then once in game scroll your mouse button to the creating AI addaction. This time you should have Ten (10) AI soldiers spawn/created in your game, all at your location. They probably all got created very quickly and probably looked like they were created all at once. Well to see how this works in slow motion, escape back to the editor, Alt-Tab back to the Ifstatement.sqf. By adding a single line to the code we can halt momentarily the loop. I invite you all to look at the wiki page for the command 'sleep' - http://community.bistudio.com/wiki/sleep - "Suspend execution for given time in seconds. The sleep precision is given by a framerate, the delay given is the minimal delay expected."

Simply add an integer or float value (i.e decimal usage) after the word sleep. E.g sleep 9.5;
Add this to our ifstatement.sqf so it should look like this...

```
 for [{_x=1},{_x<=10},{_x=_x+1}] do
{
  hint "spawning new soldier";
  _newSoldier = "B_Soldier_GL_F" createVehicle (position player);
  sleep 2;
  hint format ["Cycle: %1 Complete",_x];
  sleep 2;
 };
hint "FOR LOOP COMPLETE";
```

Save the file, Alt-tab back into the editor and then preview. Once in game, press the key to get third person, then scroll your mouse and activate the add action. See how a soldier spawns at an interval of 4 seconds around you, whilst the hints are leading you through the loop. The Cycle: %1 Complete hint will show you what loop sequence you are on. Remember – when using hint Format '%1' gets replaced with the variable _x. As _x is equal to 0 at the start you should have a sequence of 0 to 10 shown in the hint screen. Once _x has been incremented/increased to 11, and the loop has there for cycled around 11 times, it will realise now that _x is > 10 and therefore not execute the code within the do '{}' brackets. It simply exits the loop and hints FOR LOOP COMPLETE – as this is outside our loop, and is the next piece of code in the script.

With this sleep command you can see how it slows the code down so you read the hints and also see the soldiers spawn individually. Without the sleeps, you would only see the last hint and all soldiers created almost as one.

So thus far you should now have a principle concept of the 'for-loop'. Clearly you can change the condition to '< 10', or '==10', or '<2', or '< 100000'. It's a condition that defines how many times around you want to loop. But remember to increment the loop. It doesn't necessarily have to be _x = _x +1, it could be _x = _x + 24, but clearly that becomes complicated with keeping track of how many time you need to execute the code in the do brackets.

On a side note you could just as well loop backward – setting _x = 10 and then have the condition to be _x >= 0, and remove one each cycle of the loop by using _x = _x – 1;

The for-loop can be used to find out what lies within an array. An example using the 'for-loop' with arrays would be for example… say you had an array of player names and you wanted to check what they were…

```
_names = [“dave”, “eric”, “paul”, “mikie”];
for [{_x=0},{_x<=3},{_x=_x+1}] do
{
 hint format [“Cycle: %1 Complete”, _names select _x];
};
hint “FOR LOOP COMPLETE”;
```

*Remember the '%1' indicate the variable that needs to appear in the hint. _name is the variable to choose from and select indicates that you want to choose from the variable '_names'. _name select _x - on the first loop will equal 0. Which is position 0 within the array – which therefore is equal to the string variable dave. Next loop increments to 1, which selecting position 1 in the array is eric… You get the idea without me going too much into arrays. We will leave the for-loop forspec' there and will come back to it again when reviewing Arrays in more depth.

Second For Loop Type: For Var type

```
for "_i" from 0 to 10  do{};
```

Very similar to the forspec Loop – this is simply a direct request to increment/increase from a starting point (zero (0) in this case) to a maximum number – each time carrying out the code within the 'do' brackets. '_i' is simply a local variable we created to represent a variable to hold the count. You could have changed it to _bigAssCounter, or _hackingGit. '_i' is just an easy variable to use, and is widely used as the variable letter in other coding.

```
for "_i" from 0 to 10  do
{
 hint “spawning new soldier”;
 _newSoldier = "B_Soldier_GL_F" createVehicle (position player);
 sleep 2;
 hint format [“Cycle: %1 Complete”,_i];
 sleep 2;
};
hint “FOR LOOP COMPLETE”;
```

Does exactly the same as the previous for loop, the difference is really that we do not have a condition to be met, other than the maximum number to reach. This 'for loop' simply increases each cycle and finishes on the maximum number cycle being reached. Thus Eleven (11) soldiers are created, this is because we start at 0 and end on 10. Remember '0' counts as the first loop.

While Loops

So moving on from 'for-loops', we take a loop at 'while loops'. Pretty much the same concept as the 'for-loops'. While condition is met do something within the brackets. On the start of each loop check to see if condition is still met. They are kind of like a continual if check. 'If statements' are checked once, 'while loops' are checked continually until while condition is not met. Also the type of bracket is different for the if () statement and for while {}.

Wiki sets the while loop as follows…

```
while {CONDITION} do
{
   STATEMENT;
   ...
};
```

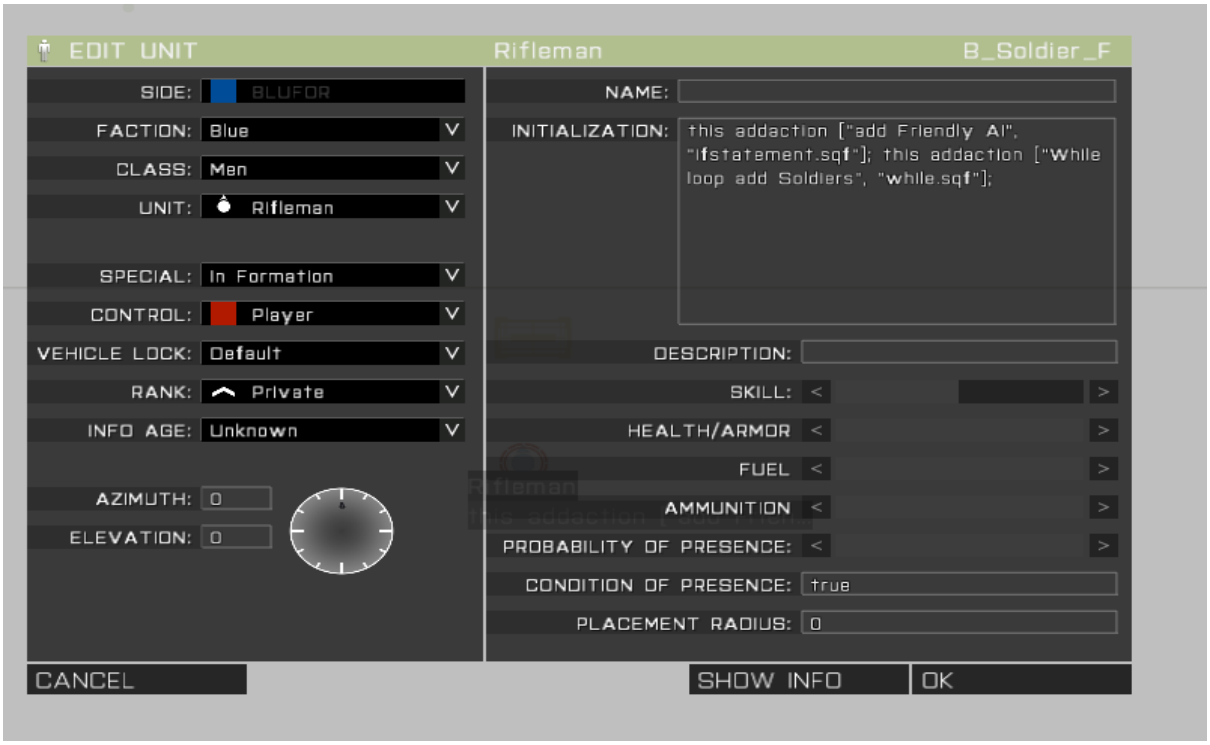
The Condition can be anything, but it must result in a Boolean. As you can see it's fairly similar in principle to the first 'for-loop' we looked at. Only difference is we are not setting any variable as the starting point within the while brackets such as '_x', and we can have anything really to be the condition that needs to be met, and not necessarily a less than (<) or more (>) than equation.
You could simply have an endless loop using {TRUE} within the brackets. This would loop endlessly, handy for when you want a script to run that constantly needs updated throughout the game, but you must use a 'sleep/waituntil' command so that it's not running full speed - continuously. THIS WOULD BE BAD!!!!

So moving up the advancement ladder, we will create another script now. Call it while.sqf.
Escape out of the game screen and into the editor for this mission, Alt-tab into a word document or Arma edit (or whatever chosen program) and create a file within your FocKTest1 mission folder called while.sqf. In this file type the following…

```
_amount = count allUnits;
While {_amount < 10} do
{
 hint "While Loop: spawning new soldier";
 _grp = creategroup west;
 _newSoldier = "B_Soldier_GL_F" createUnit [position player,_grp];
 sleep 1;
 hint format ["Units in game: %1",_amount];
 sleep 0.5;
 _amount = _amount + 1;
};
hint "While Loop COMPLETE";
```

Here we are using the correct way to spawn an AI soldier using createunit. I'll explain this in a moment after you test the script. Alt-Tab back to your game and double click on our player to access

his initialization box. Add another addaction to him using the following code, so that it looks like this…



Click ok, then click preview. Scroll your mouse button to the newly created Addaction “while loop add soldiers”, not the one directing us towards the ifstatement.sqf, then activate. This time Nine (9) soldiers should have been created around your player. If you scroll to the same addaction once again and try it this time you will just get the hint “While Loop COMPLETE”. Now for the explaining part.

Escape out of the game and back into the editor. Alt-tab to back to the while.sqf. Add the lines in red to our script…

```
_amount = count allUnits;
hint format ["Total Units in game: %1",_amount];
sleep 2;

While {_amount < 10} do
{
  hint "While Loop: spawning new soldier";
 _grp = creategroup west;
 _newSoldier = "B_Soldier_GL_F" createUnit [position player,_grp];
 sleep 1;
 hint format ["Units in game: %1",_amount];
 sleep 0.5;
 _amount = _amount + 1;
 };
hint "While Loop COMPLETE";
```

Save the script and Alt-tab back into the game editor. Now preview the game and do the same thing. This time at the very start you will get a count detailing the total number of players. It should be one (1), which is the count for our player. Then nine (9) soldiers should spawn and it will show complete. If you scroll to the addaction and activate it again, you will see that the total is 10 of players, and then straight to the hint - While Loop COMPLETE.

Escape back to the editor, Alt-Tab back to our while.sqf, and have a look at the code. I will break it down as we go along.

```
_amount = count allUnits;
```

'_amount' is a local variable as you should know by now, to which we assign the following code…

```
count allUnits;
```

'count' is a BIS command (http://community.bistudio.com/wiki/count) that's returns one of three things. 1. Number of elements in array 2. Number of elements in array for which given condition is true, Or 3. Number of sub-entries in a config object.
Hold that thought….

'allUnits' is a BIS command (http://community.bistudio.com/wiki/allUnits) that returns a list of all units (all persons except agents). Dead units and units waiting to respawn are excluded.

Looking at both together, allunits returns an array of all the players and units that are alive, and count returns number of elements in an array – in this case the 'alive players' returned from allunits. Therefore we are asking the computer to make a list of all units in the game, and then count them. This in turn returns with a value, which we assign to the variable _amount. We then use the variable to be part of the while condition that needs to be met. While _amount is less than 10 do…

So you can see why I changed to use 'createunits', I needed the computer to create non-empty units, or units that are deemed by the computer to be alive. The use of createvehicle makes dummies or

empty shells of the soldiers. Ideally this is used for creating vehicles for the game or generic objects. But using the createvehicle method is sound way to get Enemy soldiers into the game that are non-responsive and won't shoot at you -Testing purposes basically. But remember they are defined as NOT alive.

So moving on to explain the '_grp' line.

```
_grp = creategroup west;
```

Creategroup is another BIS command (http://community.bistudio.com/wiki/createGroup) that assists us in creating a new AI group (no actual soldiers – just the group we are going to place the AI soldiers in) for the given Side (west, east, friendly, etc).

_grp is a local variable we made that we will use to reference our newly created group. We are making a blank West group (the game engine gives it a name but its empty) and placing that blank group inside the variable '_grp'. When we need to reference this group we have made – we just use '_grp'. Thus when we get to the command createunit – (http://community.bistudio.com/wiki/createUnit) it states in the wiki that "…Group parameter **MUST** be an existing group or the unit won't be created." Therefore we needed to make a group so that we could create a unit in the first place. We could have just added him to the player's group. As every individual created is in a group, be it a large group or all in their own individual group, is in a group. So from the Wiki we can see something similar to the below…

"SoldierType" createUnit [position, group];

We therefore gave the type required, gave it the position of the player, and then gave it the empty group of _grp. Because the _grp variable is inside the loop it is also getting over ridden each time, much like the _newSoldier variable. So in essence on every cycle of the while loop we are creating a new group to which the solider is being placed. Thus all the units we create are in their own group, not one big one.

*Warning the game can only handle something like 250 groups in total, but can handle a shit load of AI. So don't go creating too many single AI soldiers/units in their own group otherwise you will max it out.

Now we will create units within one group just to see how it works. It's pretty much the same code, but look where the _grp line is placed. It is outside of the loop, so it only gets read once and then that's it. It doesn't change whilst that loop is going around. Each solider is being created within that one defined variable, which in turn represents our created group for the west side.

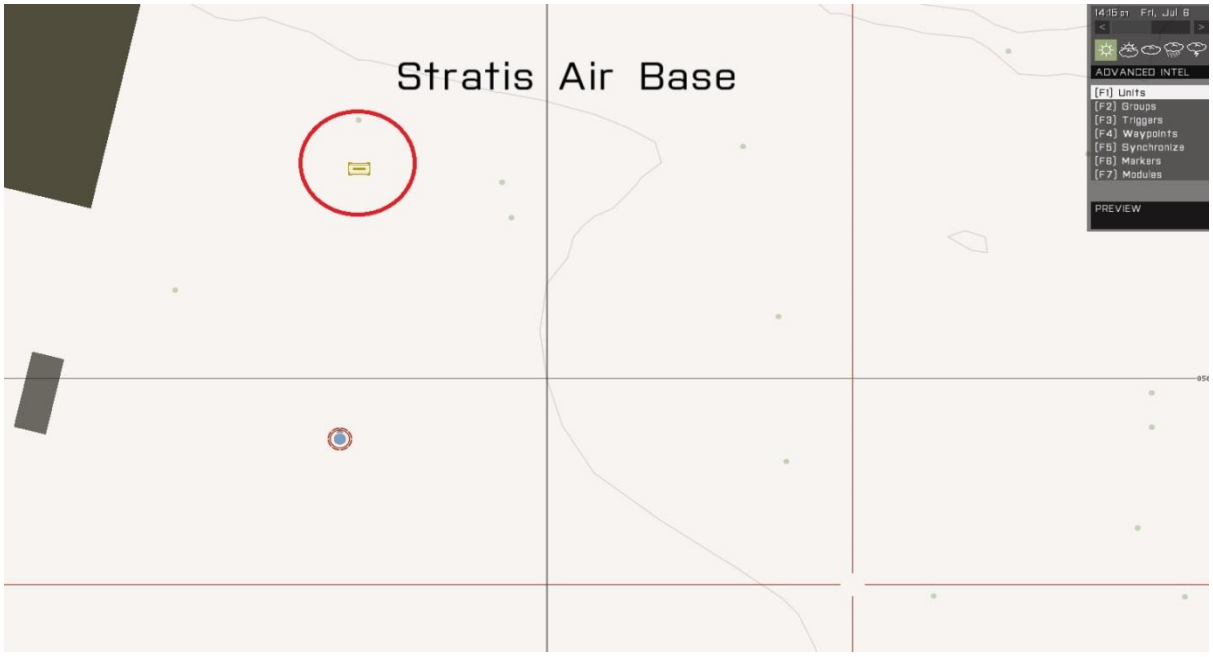
```
_amount = count allUnits;
hint format ["Total Units in game: %1",_amount];
sleep 2;
_grp = creategroup west;
```

```
While {_amount < 10} do
{
  hint "While Loop: spawning new soldier";
 _newSoldier = "B_Soldier_GL_F" createUnit [position player,_grp];
 sleep 1;
 hint format ["Units in game: %1",_amount];
 sleep 0.5;
 _amount = _amount + 1;
};
hint "While Loop COMPLETE";
```

Save the file, alt-tab back into the editor and then hit preview. Run the same addaction again and watch what happens. As the soldiers are created they all line up in a V shape around the first created unit. He is in essence the leader. Thus a big single group has been created. As we have used the createunit command instead of the createvehicle command, you can see that the soldiers are somewhat alive. They automatically line themselves up behind one another, and will now react to enemy soldiers.

Moving on…

Let's look at how the else we can use a while loop using what we have got. Escape back to the editor and move the ammo box a bit further away from the player, something similar to that shown below.



Now save the game, either preview or click save, remain in the editor then alt-tab back to our while.sqf.

Create a new file called whileDistance.sqf and add the following (new commands in red)…

```
while {true} do
{
	waituntil {((player distance AmmoSups) <= 10);};

	hint "You have Entered the Ammo Supply Zone";;

	waituntil {((player distance AmmoSups) > 10);};

	hint "You have left the Ammo Supply Zone";
};
```

Save your file and open up your init.sqf. Once inside add the line…

```
execVM “whileDistance.sqf”;
```

This is again to call the whiledistance script as soon as the game starts.
Save the init.sqf and Alt-Tab into the game editor. Click preview. Now start running towards the newly moved ammo box. When you get within Ten (10) metres of it a hint appears saying you have entered the Ammo Supply zone. Now run back greater than Ten (10) metres away from the ammo box and you should get another message saying you have left the Ammo Supply zone. You can keep doing this till your heart’s content; bit of the ‘hokey cokey’ never hurt anyone.
So breaking down each point…

`while {true} do` - this is simply a while loop in which the condition true has to be met. As a basic principle to remember – when you place true inside the while loop such as this, it basically means an endless loop. This kind of loop is handy to have running for the duration of a game, checking stuff or updating points or something similar. So the TRUE part means we have an endless loop. Got it!

```
waituntil {((player distance AmmoSups) <= 10);};
```

Waituntil – another BIS command (http://community.bistudio.com/wiki/waitUntil) which on the Wiki states “Suspend execution of function or SQF based script until condition is satisfied.”

Here we have that word condition again, and that it has to be met.

Well… ‘waituntil conditions’ can be thought of as they sound. Wait until shit happens before doing anything. So in principle we are waiting for something to happen before we move on. But what are we waiting for. Next part to consider is distance (http://community.bistudio.com/wiki/distance). The wiki states “Returns the distance in metres between two Objects, Positions or Locations.”

The syntax that it requires is object/position/location **distance** object/position/location. Well we have used the player as the left hand side object …

```
Player distance
```

Then we have used the object AmmoSups (ammo box) as the second value. So have now have…

```
Player distance AmmoSups
```

This is therefore looking to get the distance the player is away from the Ammobox (AmmoSups).

((Player distance AmmoSups) > 10)

We then enclose player distance Ammosups within brackets so that we can test the returned distance against the '> 10' part. *The 10 represents 10 metres within the Arma game engine, which equates to a radius around the ammo box. We then enclose the whole thing within brackets to ensure this is one whole element to be tested. We are therefore waiting until our player is less than Ten (10) metres away from the ammo box. Until this time the code is paused or halted within this particular script.

As soon as this 'waituntil' condition is met, the computer moves on to the next line of code. In this case it's the hint part stating we have entered in the Ammo supply zone. The code moves on from the hint to the next 'waituntil' line of code. This part of code does the opposite to the above. Its waiting until the player's distance is greater than Ten (10) metres away from the Ammo box. Until the player's distance is greater than Ten (10) metres away from the ammo box then it will wait, and the line of code halts. You will then see as soon as we run away from the ammo box, at a distance greater than Ten (10) metres, the hint appears on our screen saying we have left the Ammo supply zone. As this is a continual loop, with no condition to be met other than TRUE, then it will loop forever. Thus, the loop starts again and it waits until the player's distance is less than or equal to Ten (10) metres away from the ammo box.

You can think of the 'waituntil' command as a sort of trigger. Once that trigger is activated then continue with the code.

Just to show how else the Distance command works – if you had…

((Player distance (getpos AmmoSups) > 10)) then it is testing the distance from the player object against the ammo box's [X,Y, and Z] coordinates/position. The reason for this is the 'getpos' (http://community.bistudio.com/wiki/getPos) command, which returns the position of the ammo box's coordinates. And as we have read, the distance command can check against object, position, or location. Just something extra I thought.

END OF CHAPTER 1

Reflections… Hopefully at this point you should have grasped the basic concept of…

Variables - how we assign and initiate the variables, local and global variables, overwriting variables, the types of values we can place within a variable, and the difference between a named object in Arma and a variable.

Basic Script interaction - Creating and calling Addactions, calling scripts from the init.sqf, calling the one script from two different locations, and using one script to be accessed by different scripts.

Basic debugging – Hinting and hint format using %1 and variables.

Conditions – using the conditions that need to be met whilst looping, waiting, and if statements.

I hope this has helped you guys somewhat. Next chapter will move on to slightly more advanced stuff, utilising functions, switch command, arrays, and slightly more complex coding.

Go and make a few little scripts utilising what you have learnt, start small and work your way up. Use the wiki and the forums for guidance. If in doubt search, and if that doesn't work, and I mean a proper search, then ask for help. The community are really helpful, but like most they won't do it for you. The best way to learn is fecking around yourself and seeing how things work. Use hints to break up your scripts, and use waituntils/sleeps to slow things down.

Anyone wanting assistance with this tutorial please email me at fockersteam@hotmail.co.uk – happy to help. Anyone with suggestion please do the same.

Refs

http://en.wikipedia.org/wiki/Variable_(computer_science)

http://community.bistudio.com/wiki/Main_Page

Show Script Error, and Window mode

Two hand tools is to have the game engine running in a window mode to allow you to alt-tab easily and secondly a message system that will show you when you have errors. All you do is go to your steam folder and find within that the Arma 3 folder. Once inside you should see the contents of that folder and should see the executable/application icon for Arma 3. You need to make a short cut for that icon/send to desk to, by using the right mouse button on it.

Once the Icon is on the desktop, right click on it, select properties. If you have only got 5 tabs within this property page then you have not created a SHORTCUT. Go back and do it again. If you have then in the shortcut tab, click in the Target: box and right at the end of that after everything else in there (DO NOT DELETE ANYTHING) ass this line as shown…

-nosplash -showscripterrors -window

Save it and then use this Icon to run the Arma game when editing.