5/19/2013

[FOCK]ERS

FOCKERS ARMA 3 SCRIPTING GUIDE – CHAP2

*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.

*Massive thanks for all the positive feedback I received for chapter one – really appreciated – I will place all chapters together on completion, and when correctly edited, I will release one big tutorial.

Regards

Mikie J [FOCK] – BIS forums Mikie boy

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

CHAPTER 2 – BASICS And Beyond

Part 1 – Array Basics

In the last chapter we touched upon Arrays…

_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"]. "

So what is an Array – well it's certainly not something that comes out of a futuristic gun…. Best way to think of them is as a list of stuff, which is held within a variable. By stuff I mean – list of objects, locations, markers, variables, strings, integers, etc. – like I said "STUFF".

From the above you can see you declare and array in the same manner you would declare a string or integer variable. However you are providing the computer with a little more info, that being the '[]' brackets which encases that which you wish to place in the array.

Let's face it men don't do lists…they rather remember shit off the top of their heads and hopefully that will suffice. Well I for one am happy to accept that the wife is right and her almighty shopping list does the trick! I can hear a 100 men sigh and mutter "bollocks" at the same time, lol. Thus, the case for arrays is a well-placed argument, and an effective means to shorten what we have to remember, and in some many words… what the computer has to remember.

For-example, what would be easier to remember?

```
_aiDude = Dave;  //oh yes he's back!
_aiDude1 = Eric;
_aiDude2 = Paul;
_aiDude3 = Tom;
_aiDude4 = Dick;
_aiDude5 = Harry;
```

Or

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];
```

So that's that argument put to bed. It's therefore self-explanatory why we would use them. Well now onto how we use them.

Array initialization

Initializing arrays can be done manually by simply following that which I demonstrated above. Give the Array a variable name (local or global) and then initialize that array variable with an '=' sign and provide '[]' to enclose any data inside. The data inside is referred to as 'elements'. Arrays in Arma

can hold an infinite amount of elements as far as I can tell, of various types, mixed if need be, but we will start off with simple stuff.

In your FockTest1 mission folder, create a new file called Array.sqf. Inside this script type the following…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];

Hint format ["%1 ",_aiDudes];
```

Now goto you init.sqf, and add the Array.sqf to run on the game starting…

So inside init.sqf type…

```
execVM “Array.sqf”;
```

Save the file. If Arma is not open – do so now and load into your FockTest1 mission. In the editor simply press preview and see what happens…

Yep – almost Feck and all! But least you got some brackets and six (6) ‘any’ words displayed inside, separated by a comma. So what’s does that mean?

Firstly you can see we used hint format to get the array to be shown. Hint _aiDudes will not work! Try it and see, and if you have showscripterrors on you will see an error stating it requires text. Thus this method does not work. By Placing the variable array within the format function you place “ “ either side of the brackets to allow it to be shown. All it is doing is making the array, as a whole, a string. You can’t do anything with the array in this form but at least you can see it. As for the ‘ANY’ shown in the text, well these relate to all the names you placed inside the array. They are showing as ‘Any’ as they are deemed to be objects which are presently non-existent, particularly as they are not each encased within “” such as [“Dave”, “Eric”, “Paul”, “Tom”, “Dick”, “Harry”];

The array technically is made up of game objects with the names Dave, Eric, etc inside – all of which we don't actually have in the game. If however you placed those objects in the game and name them accordingly, what would happen. Try it, making sure you have removed the names in the array from " " – should look like _aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];

Save the file, Alt-Tab out of the Array.sqf, and back into your editor. Place six (6) individual blufor soldiers (non-playable) individually on the map and NOT WITHIN the group of your player (press F2 to un-link the soldiers from your player if they are linked) and NOT within the same group as each other. Name the blufor soldiers accordingly (Double click them and type in the name line) – Dave, Eric, Paul, Tom, Dick, Harry. Click preview and see what hint you get…

Now you have the array shown as before but this time the names are displayed instead of 'ANY'. Why? Well these names represent actual game objects. So we now know the array holds names that are actual objects in the game.

That's how we therefore make a simple array, but how do we select a specific element from that array, namely one of the Named Objects.

Selecting Array Elements

As we created the Array manually we can see that Dave is placed in position 0, Eric is in position 1, Paul in position 2, etc. Well that makes it easier knowing said object's position. So, let's select Dave and get him to do something.

Exit from the game back to the editor, alt-tab back into the Array.sqf. Change the line… Hint format ["%1 ",_aiDudes]; … to … Hint format ["%1 ",_aiDudes select 0]; Save and alt-tab back into the game editor. Click preview and see what the hint shows. Yep it only shows the name Dave. So what happened there? Hopefully you will remember me waffling on about the magic '_this' when dealing with addactions. I said '_this' acted/represented (in this particular circumstances) as an array, and from that array we selected the target, caller, or ID, by adding select and the position in the array. Well that's the same way we generally select elements from and array.

In this newest example we simply said – here's an array called _aiDudes (containing our Named game objects) and I would like to select the element located at position 0, and then hint that particular element on the screen…

```
 [Array]     select command       Position within the name array.
_aiDudes        select                          0
```

If we wanted to choose Eric instead of Dave, we would declared the array _aiDudes, then use the command Select and then provide Eric's position within that particular Array. In this case Eric is located at position 1. Thus…

```
_aiDudes select 1
```

Simple, declare the array, use the command select (http://community.bistudio.com/wiki/select) and then the desired object/string/value's position within the declared array. Try selecting Tom and Harry separately, and then hinting them one after the other. Your basic code should look like this…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];

Hint format ["%1 ",_aiDudes select 3];

Sleep 1.5;

Hint format ["%1 ",_aiDudes select 5];
```

Moving onwards. Hopefully now you should feel that you are getting somewhere and hopefully the scripting is becoming easier, both to understand but also to use and do yourself without guidance. It's at this point I would invite you to practice dicking around with the simple stuff I'm giving you. It's the only real way to learn. Simply following this tutorial will only give you some of the knowledge to move on, experience however is the key to developing.

So how would you randomly select someone from the _aiDudes array and hint the on the screen? Well we know the array is constructed with each element having been given its position - 0,1,2,3,4,5. And we know how to select a position, so all we have to do is randomly select a number between 0 + 5, then that selected number can be placed after the select command. The best way to do this is to create a variable called _rnd. This will hold our randomly chosen number between 0 and 5 and can be placed after the select command as such…

```
                        THE ARRAY  - SELECT -  POS IN ARRAY
Hint format [" %1 ",_aiDudes select _rnd];
```

In this position it will now act as the selected position within our _aiDude array. All we have to do now is create the random number to place into the variable _rnd. To create a random number we can use the BIS command Random (http://community.bistudio.com/wiki/random). If you look at the wiki it states that it will return "Random real (floating point) value from 0 (inclusive) to x (not inclusive)". Notice how it states floating point, which is a two-placed decimal number E.g 2.01. So this means requesting a random number will not generally give us a whole number. Thus we will add the command floor (http://community.bistudio.com/wiki/floor) in front of our Random command in order to obtain a whole number, as the Floor wiki states "Returns the next lowest integer in relation to x." Basically it rounds down float numbers to whole integers (non-decimal numbers). So we use the following code in order to return 0,1,2,3 ,4, or 5…

```
_rnd = floor (random 6);
```

Six (6) is placed after the BIS command random. Six (6) is therefore the number by which the random number will work from, Zero (0) to six (6). Six (6) is used as there are six (6) elements in our array. Random then picks a float number between Zero (0) and Six (6), which the floor command then rounds down to the nearest integer, thus six (6) is never chosen. We only need Zero (0) to Five (5) as our array only holds those positions (0,1,2,3,4,5), thus this is why we use six (6). Therefore we declare our variable _rnd, which we then initialize with the returned result from our floor random number. Change our code to reflect this…

```
_rnd = floor (random 6);

_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];

Hint format ["%1 ",_aiDudes select _rnd];
```

Save the Array.sqf, Alt-Tab back into the game editor, preview the game and hopefully now a random object has been selected from that array. Escape out of the game back to the editor, Alt-tab back into the Array.sqf.

Now we know how to use the random command, let's randomly select more than one AI Soldier and then hinting them one after the other. This is where our for-loop knowledge will come in handy. We need a loop in order to randomly select a different number (EACH TIME) that we can use to select a position from our array. Change the code once more to reflect this…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];

for "_i" from 0 to 1  do
{
_rnd = floor (random 6);

Hint format ["%1 ",_aiDudes select _rnd];

Sleep 2.5;

Hint “”;

Sleep 1;
};
```

To ensure the code doesn't run until we are in game, and so that we don't miss the hints, open the init.sqf file and delete/comment-out (using '//') the line execVM "Array.sqf"; Save the file and go back into the Array.sqf. Save that file, Alt-tab back into the game editor, double click on your player to access his init box. Add the following line…

```
this addaction [“Rnd Array Select”, “Array.sqf”];
```

..So that we can call the Array script ourselves as many times as we want.
Click preview, scroll your mouse to the 'Rnd Array Select' addaction – click on it and see what hints this time.

Hopefully you should have had two hints on your screen, and hopefully they are different. There may be likelihood that you got the same name twice. If you did not, call the addaction again. This time you should get different names.

Breaking down the code…

Well we know how the for-loop works and we know how the random command works, and we know that the variable _rnd is equivalent to our random value (0-5). Therefore each time the loop goes around, it will select a random number and place that number after the select command within the hint format line. This in turn selects the AI Soldier at that position within the _aiDudes array. This AI Soldier is then hinted on screen. If however we placed the _rnd variable outside of our for-loop scope, instead of inside it; when the script runs it will randomly choose a number and place that value inside of _rnd. When the code moves onto the for-loop, it will run twice but will use the same number defined by _rnd, as it is not being renewed/overwritten on each cycle – as it is not being called twice.

Finally on this little bit…

```
Sleep 2.5;

Hint “”;

Sleep 1;
```

I added these three lines – to allow the hints stay longer on screen just in case you got the same name twice. Sleep as we know pauses the script, which in this case allows the hint to stay on the screen longer, then hint “” is simply a blank hint to remove the last hint on screen (of the chosen object) that would have been on the screen after the first loop. The final sleep is to allow for that change in hints to be noticed.

Ok, so hinting is getting fecking boring, but like I said previously, it’s a good method to use as a debugging tool.

Let move on to something more interesting using the random select and the for-loop array.

Escape out of the game back to the editor, Alt-tab out of editor back to our Array.sqf. And amend the code to resemble the following…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];

for "_i" from 0 to 1  do
{
_rnd = floor (random 6);

Hint format ["%1 ",_aiDudes select _rnd];

Sleep 4;

(_aiDudes select _rnd) doMove (position AmmoSups);

};
```

Hopefully, you should see either the one same name stay on the screen for a long time and one soldier move to the ammo box, or two different names appear on the screen and two soldiers move to the ammo box as shown below.

Awesome!! Now we are actually doing something of use - Moving soldiers around and less hinting. Quick look at domove Command (http://community.bistudio.com/wiki/doMove)...

"Order the given unit(s) to move to the given position (without radio messages). After reaching his destination, the unit will immediately return to formation (if in a group); or order his group to form around his new position (if a group leader)."

You can see the reason why we ungrouped the units, so they don't return to formation.
The Wiki states that it requires a unit (either in an array format (for more than one soldier) or object format (for a single unit)) and a position. Position in this case is defined with position AmmoSups, which returns the X,Y, and Z coordinates for the ammo box.

Basic Subtracting and adding to Arrays

Before we finish with hinting a millionth time, I will just briefly explain the basic concept of adding to and subtracting from arrays. This level of array manipulation, as I have said, is basic but it will allow you to add to arrays and subtract from them when required.

Basic Subtracting

*I will talk about using the set command in advanced arrays

As it stands using the code above, selecting and moving soldiers, has one issue; that is, how we actually select the soldiers. Yes it works, but as I have said the same soldier can be chosen twice, which is not what we want, especially when using random. So let's fix this…

In the next bit of code I have added a few more hints than normal, but it's just to show you how each part of the code is working on screen, and to assist you visualise what's happening. Otherwise half of you will simply copy the code, get it working, and feel happy not knowing, while the other half will sit there trying to work out what the feck just happened. Also this is a useful for practicing

with the use of hint format. I've highlighted the hint and sleep in red so you can actually see how small this code is, and what to delete once you have understood what's happened.

So escape back to the game editor, Alt-Tab back to the Array.sqf and remove the code in place. Now type the following…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];

for "_i" from 0 to 5  do
{
	_total = (count _aiDudes) - 1;
	hint format ["Array Total: %1",_total];
	sleep 2;

	_rnd = floor (random _total);
	hint format ["Random No. %1",_rnd];
	sleep 2;

	_pick = _aiDudes select _rnd;
	hint format ["Soldier Picked: %1",_pick];
	sleep 2;

	(_pick) doMove (position AmmoSups);

	_aiDudes = _aiDudes - [_pick];
	hint format ["Original Array: %1",_aiDudes];
	sleep 2;
};
```

Save the Array.sqf, Alt-Tab back to the game editor and press preview. Scroll your mouse button to select the 'rnd Array Select' addaction and active it. Watch how the code is broken down through the hints, selecting a random player from the array list then moving that soldier to the ammo box. How'd that look? Pretty awesome right… Really simple, but all the AI soldiers moved randomly to the ammo box. If you restart the mission over and do it again, see how whom it selects is random each time. Escape back to the editor and Alt-Tab back to the code.

Breaking the code down…

`_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];`
Well this hasn't changed, still the original array we started with – which holds all our named AI soldiers that we have placed in the game.

`for "_i" from 0 to 5  do`
Because we now want to select every member in our array we have changed the one (1) to five (5), which is the number of AI soldiers in our array.

`_total = (count _aiDudes) - 1;`
We make a new local variable called _total. We initialise it with the code (count _aiDudes) – 1. We already know what count does, so we can see it counts the total of items in the array named _aiDudes. Then outside of the bracket we minus one (1) from the total number returned from the count command. So, whatever the total that is counted for units left in the array (named _aiDudes),

we minus one (1). This is because if the total count of AI soldiers is six (6) and we select that number in the array, we would get an error. This is because there is nothing in position six (6) in the array. REMEMBER!!!! Array positions start at Zero (0). So when we get down to one (1) unit left in the array, his position in the array is at position Zero (0), Not one (1).

_rnd = floor (random _total);
We make a new local variable called _rnd. We initialise it with the code floor (random _total). We now know that floor returns a whole number, and we know that random command obtains a random float (decimal) number from any number that follows it in the code. In this instance _total follows the command random, which we know from above returns the total amount of AI soldiers counted in our array, minus one(1). Remember it gives us a number not any AI Soldier, just a number. Due to the brackets surrounding the command random and the variable _total, the random number from the '_total' amount is returned first. Once this is obtained then the floor command kicks in and gets the nearest whole number. It's the same as this…

```
_rnd = random _total;
_floorNo = floor _rnd.
```

You can see we get the random number first then obtain a whole number from the random total.

```
_pick = _aiDudes select _rnd;
```

We make another new local variable, this time called _pick. We initialise it with the code _aiDudes select _rnd. Here we are using the array _aiDudes that holds a list of our in game AI soldiers, and we are selecting the position in that array by using the number returned from _rnd value. Remember, _rnd stands for a random number we created by counting the amount of AI Soldiers in our array and randomly selecting a number between Zero (0) and that total amount. So simply, we are selecting a Solider at random from our array _aiDudes and then the variable _pick holds that randomly chosen AI Soldier. So from this point on any time we mention _pick, it actually stands for the chosen AI Soldier from our array.

```
(_pick) doMove (position AmmoSups);
```

As _pick now stands for the chosen AI Soldier from our array, we are telling the computer to move the chosen AI Soldier to the position returned from the ammo box named "AmmoSups".

```
_aiDudes = _aiDudes - [_pick];
```

Now we have identified a Soldier from our array (_aiDudes) we are giving the array a new value. We are asking the computer to remove from our array the AI soldier we have randomly picked. We do this with the _aiDudes – [_pick]. The brackets surrounding our chosen AI soldier signify we have an object in an array. You cannot simply put _aiDudes - _pick, as this would be trying to remove an object from an array which returns an error. Although that is exactly what we are trying to do, you can only remove an array element from another array. This is done by placing the array elements you want to remove inside an array (square brackets) or create a whole new array altogether. Then use the minus sign to do the subtraction. Look at the following an example…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];
_aiDudes1 = [Dave,Tom];
_aiDudes = _aiDudes - _aiDudes1;
```

Or

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];
_aiDudes = _aiDudes - [Dave,Tom];
```

Both return an array containing [Eric, Paul, Dick, Harry]. You can therefore see the _aiDudes array has had the two specific elements removed from it.
Once the first cycle of the 'for loop' goes round, you can see that one AI soldier has been removed from the array. On the second loop of the 'for-loop' we now have one less in the array to deal with. This is why we use the count command on each cycle to ensure that we get the total amount each time for the array, and not sticking with Five (5). For example, if we got to the third cycle of the 'for-loop' and wanted a random number between Zero (0) and Five (5), then if we selected Five (5), there would not be anything at position Five (5) as they have been removed. Thus counting the total each time ensures we have a correct maximum number (of the array) to work with.

Therefore you should see that as the loop goes round, the computer randomly selects an AI Solider from our array, who then gets removed from the array and told to move to the ammo box. Then as the loop goes around, our array of AI Soldiers gets smaller until all have moved to the ammo box and all have been removed from the array _aiDudes. You can see this each loop when the hint appears showing the 'Original Array' contents.

Adding

Pretty much the same method as subtracting, but obviously we are going to use the addition command.

Ammend the Array.sqf to show the following…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];
_newAiDudes = [];

for "_i" from 0 to 5  do
{
	_total = (count _aiDudes) - 1;
	hint format ["Array Total: %1",_total];
	sleep 2;

	_rnd = floor (random _total);
	hint format ["Random No. %1",_rnd];
	sleep 2;

	_pick = _aiDudes select _rnd;
	hint format ["Soldier Picked: %1",_pick];
	sleep 2;

	(_pick) doMove (position AmmoSups);

	_aiDudes = _aiDudes - [_pick];
	hint format ["Original Array: %1",_aiDudes];
	sleep 2;

	_newAiDudes =  _newAiDudes + [_pick];
};

hint format ["New Array: %1",_newAiDudes];
sleep 1.5;
```

```
hint format ["Old Array: %1",_aiDudes];
```

I've typed the new code typed in red to make it easier. Save the Array.sqf, Alt-Tab back into the editor, then click preview. In game use the addaction to access the Array.sqf. Watch again as the soldiers move to the ammo box, and follow the on screen hints. But this time watch for the 'New array' and the 'Old array' hints at the end of the script.

You should see that our original array is now empty, denoted on screen with 'Old Array []', and the new array hint shows all our AI Soldiers.

We have therefore deleted all elements from one array and then added elements to a totally new array.

Breaking down the code…

```
_newAiDudes = [];
```

We start the code by creating a new variable called _newAiDudes, and make this an empty array (brackets with nothing between). The loop goes round as before, each time removing the randomly chosen AI Soldier. This time however we have this line…

```
_newAiDudes =  _newAiDudes + [_pick];
```

Here we are adding to our empty array. We are placing our randomly chosen AI soldier inside an array and then adding him to the new empty array. Each time the loop goes around the array gets bigger. Remember, because we are saying _newAiDudes is equal to _newAiDudes plus [array element]; we are saying "keep the original contents of that array but add one extra element onto the end." Same concept as _i = _i + 1;

Now we have a new array called _newAiDudes which holds all our AI soldiers. Let's use the code to call them all back to us once they have all reached the Ammo box. This time however, when we call the soldiers back to us we will use a new command instead of a 'for-loop'.

forEach loops

Let's start by looking at the command foreach (http://community.bistudio.com/wiki/forEach).

Wiki states… "Executes the given command(s) on every item of an array. The array items are represented by _x. The array indices are represented by _forEachIndex."

Hmmmmmm? Don't panic! It's a really straight forward command as long as you have understood the arrays so far. Escape out of the game back into the editor, Alt-Tab into the Array.sqf and add the following code to our Array.sqf after the last line…

```
{

   (_x) doMove (position player);

} foreach _newAiDudes;
```

Righty, "this looks fecking strange" I bet your telling yourself. Well, make sure you haven't altered the code from above and left the sleep commands in place. Save the Array.sqf, alt-tab into the editor, click preview and run the game. Same process again – call the add action for the Array.sqf and see what happens.

Yep, once they all get to the ammo box, they all return to you together. So what happened to make them do that?

Well think of it as this… "for each element in our array (the new created array in this case), execute the following code. Where _x appears in the code replace it with one of our elements". Therefore where _x appears, it tells the computer to replace it with an element from our array – in this case an AI soldier. It then executes the code for that particular AI Soldier, and then it moves onto the next AI Solider (element on the array) and executes the code for them… This continues until all elements in the array have had the code executed upon them. It's like a for-loop, but its carrying out a specific function at the same time, jumping from one element of an array to the next, carrying out the necessary code.

Thus, it starts at position Zero (0) in our chosen array and then moves upwards until all elements have had the code executed for them.

You can add a sleep command to slow it down if need be; much like that of a 'for-loop'.

```
{
  (_x) doMove (position player);

  Sleep 2;
} foreach _newAiDudes;
```

NOTE – if you remove all the sleep commands from the original code (sending AI to Ammo box) they will all move a few yards and then come back to you. This is because the code is being executed extremely fast. If you want to remove the sleep commands and hints – you will need to use waituntil command…

```
{
waituntil {(((position _x) distance (position AmmoSups)) < 8)};
} foreach _newAiDudes;
```

Hopefully you remember how the waituntil command works, and also that of the distance command. This little piece of code basically states…wait until each element (AI Soldier) in our array (_newAiDudes) is less than Eight (8) metres away from the ammo box named AmmoSups. Once they are all within Seven (7) metres of the ammo box execute the next bit of code – in this case return to the player.

You should now see how handy this forEach command is. If not – you're a tool!

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];
_newAiDudes = [];

for "_i" from 0 to 5  do
{
	_total = (count _aiDudes) - 1;
	_rnd = floor (random _total);
	_pick = _aiDudes select _rnd;
	hint format ["Soldier Picked: %1",_pick];
	sleep 2;
	(_pick) doMove (position AmmoSups);
	_aiDudes = _aiDudes - [_pick];
	_newAiDudes =  _newAiDudes + [_pick];
};

{
  waituntil {(((position _x) distance (position AmmoSups)) < 8)};
  (_x) doMove (position player);
} foreach _newAiDudes;
```

You can also use multiple commands in the same block of code – that being anything between the curly brackets {}. Just think logically how you use the brackets. See how I've changed the last piece of code removing one of the foreach commands. I've wrapped the two bits of code into one block of code. Thus you cannot use the foreach command without wrapping what you want to affect within curly brackets (I really should look up the proper word for the brackets!!).

If you want to know more now about arrays, please look at this helpful link from the community (http://forums.bistudio.com/showthread.php?100559-Beginners-guide-Arrays), and also the wiki page located here (http://community.bistudio.com/wiki/Array).

Part 2 – Function Basics

Local Functions

What is a function? Well as it happens the wiki (http://community.bistudio.com/wiki/function) sums it up nicely…

"A **function** is a piece of code which performs a specific task and is relatively independent of the remaining code. They often accept input parameters and sometimes return values back to the script that called them."

"Functions should be used for any processes where the **result** or **calculation** done in the function is important. This result or calculation should be made in the **least time possible**. They are unlike scripts, where *timing* is important.

Functions can also be used to **reuse code**. You can write some code once in the function and then include it in many different scripts. When the code is updated, it is updated for all scripts. When you only copy and paste the code to the other scripts, you have to update every script on any change."

There is plenty of information there to give you an idea, but functions are one of those things that need to be played with in order to understand. Firstly let me explain how the function is created. Well you give the function a name as you would a variable. The function can be local, or global. All you have to do is place the code of the function inside the curly brackets {} (yes I still haven't looked up the name for these fecking things). You need to the initialize the code by the usual means of the equal sign. For example _myFunction = { code }; Any code inside of the brackets is local to that function only. You cannot directly refer to any variable you place/create inside. I will explain this later in a separate chapter. So let get cracking and create a new blank script called function.sqf. In that script type the following…

```
_hinting =
{
  hint "Function has been called";
};
call _hinting;
```

Save the file, and alt-tab back into the editor. In the editor open you your player's init box (double click on him) and add the following addaction line to your init box…

```
this addaction ["Function" , "function.sqf"];
```

Click ok and the preview. In the game scroll to the function addaction and activate it. You should now have a simple hint on the screen saying "Function has been called". So right about now you must all be thinking what's the fecking point of that, that's sucks ass… And you would be right, except this needs to be done to show you the basics, so stop bitching and get on with it! ☺

What we have done here is made a function that does not return anything to us – it just sends a hint to the screen that's says "Function has been called". However, say this was almost twelve (12) lines long and was a regular hint that needed to be called, say a regular call to this function to remind

players not to act like bell ends and crash every chopper possible! (im not whinging! Lol). Thus, you don't want to type this out a load of times in different scripts, simply calling this one function would do that for you. Although this is not the best example it's at least got you thinking.

Breaking down the code…

You see the call command – this is used to call the hinting function. In order to be able to call the function it has to be loaded into the computer's memory before the call command is used. This is what makes it faster than a script (in so many words). So in this instance when the addaction calls the function.sqf, it accesses the function.sqf and reads the hinting function into memory – storing it all in the variable _hinting. It then moves onto the call hinting part. As the hinting function is already stored it simply calls that function. Try placing the function after the call hint and see what happens…

Yep it doesn't work…Well actually nothing happens. This is because the variable _hinting hasn't been loaded into the memory and therefore the variable _hinting has no meaning at the point the call command is used.

Let's take a quick look at the command call (http://community.bistudio.com/wiki/call) and (http://community.bistudio.com/wiki/Function). If you look at the latter URL link, and scroll down to the call section, you will see the following…

"Functions executed using call are run *within* the executing instance, which waits for the result of the function. Unlike scripts, functions halt all other game engine processes until the function has completed its instructions. This means functions run faster than scripts, and the result of functions is immediate and unambiguous. It can also mean that if a function takes too long to run it will have an adverse effect on game play - large functions or CPU intensive functions can cause the game to seize up until it completes. When creating a function you want the function to be short and sweet to achieve the best results."

Both wiki pages show examples of executing a string, don't get confused by this; for now use the process if have shown you, and as we progress it will become clear.

Whilst we are talking about Call, we might as well have a quick look at the command Spawn (http://community.bistudio.com/wiki/spawn) and again (http://community.bistudio.com/wiki/Function).

Looking at the latter link once more it states…

"Functions may also be executed using spawn, but then the function result is not accessible, making it behave more like a procedure. Spawned functions will run asynchronously or *alongside* the executing instance. This helps prevent large CPU intensive functions from seizing up the game."

For now, and just for now, use spawn to execute any function that has a sleep command inside of it, and if you want the function to return a value (this is explained below).

Passing information to the function

As you look at our hinting function it's pretty pointless at present, unless we wanted to hint that small line over and over. Instead let's give our function something to do. We can pass a value/object/string

or elements from an array to the specified function. . Hopefully you will remember the addaction explanation when referring to '_this'. Remember "_this " is a magic word used by BIS coding structure, and in the usage of the 'addaction' I said "IN THIS SPECIFIC CASE" it is an array of things. Well, here it can be used as a single object/value/string or as an array. Let me explain…

Type player in front of the call _hinting so it looks like…

```
Player  call _hinting;
```

Here we are passing the player object to the function, so the hinting function can hint the name of our player. Now in the hinting function let's set it up to receive the player's object. Amend the hinting function so it matches the below…

```
_hinting =
{
_receivedValue = _this;
Hint format ["Value sent to function is: %1", _receivedValue];
};
```

Above you can see how I've initialised _receivedValue to receive the value of _this. Well in this instance _this is not an array – it represents a single object (in this case), that being the player we sent to function. We could have easily sent a string (`"whats up" call _hinting;`) to the function instead of the player, or a number (`245 call _hinting;`).
As we are not sending an array of things at the moment to the function, _this represents that one item of data, hence why we don't place _this select 0 instead of just _this. This is because we don't need to select anything from an array as there is not one, just one solitary item. Again this is our player object.

Save the function.sqf, Alt-tab back to the editor and press preview. In game scroll your mouse to the function addaction and activate it.

*I left player in the hint instead of the word value

There you have it. You're player's object name is hinted in the game. Great back to hinting our name!!! lol don't panic, I'm planning on progressing to something useful…

As our hinting function is set up to receive any one value/object/string, test it by changing the player object to a string or a number, or Dave object and see what results you get…
Once you have done that call all three to the same function as below. See how it calls the same function three times and gives a different result each time depending on what you passed to the function.

```
"whats up" call _hinting;
Sleep 2;
2231 call _hinting;
Sleep 2;
Dave call _hinting;
```

Although that is a lame use of a function, you can see how handy a function can be. We got three different items to hint on the screen by calling the same function three times. Once that function is in memory it remains until it is overwritten.

What happens if we want to call the function once, but send it the three different items as above? Well although I personally don't suggest at this stage to be mixing items in an array, we are going to do it to show you have to pass mixed array elements to a function.

Alt-Tab back to the function.sqf and in front of the call command we now place square brackets [] which constitutes an array. We should have this…

```
[] call _hinting;
```

We now need to place some information inside that array, so let's place all three items into the array…

```
["whats up", 2231, Dave] call _hinting;
```

We now need to set up the hinting function to receive the three elements we are sending it. This is where the magic word _this is utilised once again, but this time as we are sending elements of an array, and not just one item, we need to select from that array, so we use '_this select' position in the array. Position Zero (0) is "What's up", position one (1) is the value 2231, and position two (2) is the object Dave. Change the hinting function so it resembles the below…

```
_hinting =
{
_ValueOne = _this select 0;
_ValueTwo = _this select 1;
_ValueThree = _this select 2;

Hint format ["Value one: %1, Value Two: %2, Value Three: %3", _ValueOne, _ValueTwo,
_ValueThree];

};
```

Save the function.sqf, alt-tab back to the editor and click preview. Again execute the function addaction and you should get this…

There you have it, three different items passed to the array with a simple hint to highlight what was passed. Let's bump this up a little…

Imagine for whatever reason we needed to calculate three numbers all throughout our mission, over and over. Yes typing _sum = number + number + number; hint format ["%1",_sum]; Would be fine once or twice, but it would get boring if you had to do it say 30 times in a script (why you would be doing this I have no idea, but humour me).

That's where we would use our function concept. So, escape out of the game and back to the editor, Alt-tab out of editor and back to our function.sqf. Now amend our call to function part…

```
[1,2,3] call _hinting;
```

Inside I have placed three values that are to be passed to our hinting function. We must now amend the hinting function to calculate the three numbers passed to it…

```
_hinting =
{
  _ValueOne = _this select 0;
  _ValueTwo = _this select 1;
  _ValueThree = _this select 2;

  _sum =          _ValueOne + _ValueTwo + _ValueThree;
  Hint format ["Total: %1",_sum];

};
```

*Note - we can simply have hint _sum, as _sum is an integer and we can only hint strings, thus we encapsulate it within the format process to enable it to be visible onscreen as a string.

Save the function.sqf, Alt-tab back to the editor and click preview. Yep the number 6 is hinted on the screen. Thus, you have successfully passed elements from an array to a function. Now escape back to the editor, Alt-tab back to the function.sqf, and under [1,2,3] call _hinting; add a few more calls to the function with different numbers.

You must use the command sleep between each call otherwise the function hint will be so quick you won't see it until the last one, as each hint will overwrite the next. Obviously this is only necessary for displaying hints, it's not a must.

Try the following…

```
 [1,2,3] call _hinting;
Sleep 1.5;
[5,2,3] call _hinting;
Sleep 1.5;
[1,67,3] call _hinting;
Sleep 1.5;
[1,25,30] call _hinting;
```

As you can see, this is so much easier than…

```
_sum = number + number + number;
hint format [“%1”,_sum];
sleep 1.5;
_sum = number + number + number;
hint format [“%1”,_sum];
sleep 1.5;
_sum = number + number + number;
hint format [“%1”,_sum];
sleep 1.5;
_sum = number + number + number;
hint format [“%1”,_sum];
sleep 1.5;
```

Hopefully now that makes a little more sense as to why you would use such a function. Moving on to the next part of the function operating system before we actually do something useful to progress your mission.

Returning Value

Still using our hinting function, let’s assume we need to use the value from our sum of the three elements passed to the function. In the first call to the hinting function, when we passed three numbers [1,2,3] to the function, it displayed 6. Let’s say we wanted to use that total of six in our script. We would need our function, in effect, to pass the result back to the call _hinting bit so we can use the total elsewhere. To do this we would first have to create a new local variable which would be initialized with [1,2,3] call _hinting; Call this variable _total. Amend the code accordingly…

```
_total = [1,2,3] call _hinting;
Hint format [“Returned Value: %1”, _total];
```

What this means that when we get a total from the _hinting function, whatever value we return from it we place inside the variable _total. To get the _hinting function to return a value all we do is place the variable _sum on its own at the end without a semi colon after it…Also delete the hint format from the function.

```
_hinting =
{
_ValueOne = _this select 0;
_ValueTwo = _this select 1;
_ValueThree = _this select 2;
_sum = _ValueOne + _ValueTwo + _ValueThree;

_sum
};
```

So now once the call to the function has been made _total equals _sum; Save the function.sqf and alt-tab back to the game editor. Click preview and activate the function addaction. Tada!!! You have now pulled the value from the function. Again not that exciting but at least you can see what's going on. So one last thing using returned values before we move on. Amend the call to the _hinting function by using the following…

```
_total   = [1,2,3] call _hinting;
_total1 = [1,2,3] call _hinting;
_total2 = [1,2,3] call _hinting;
_totalFinal = [_total, _total1, _total2] call _hinting;
hint format ["Returned Value: %1", _totalFinal];
```

Save the function.sqf, alt-tab back to the editor and then preview the game. Again execute the function addaction and hopefully the sum on your screen should be Eighteen (18). You should be able to see that we called the same function three (3) times and each time returned a value that was stored within the calling variable. We then one last time called the _hinting function asking it to add up the value of all the returned totals. Thus, 6 + 6 + 6 = 18. Again nothing mind blowing but this is the basics of functions, and I'm sure a lot of people out there did not realise that functions can return values.

So now to this point we have looked at the basic setup of arrays, how to add and subtract from them, and how to loop through them using forEach command. We have also looked at the creation of local functions that are called within the same script, and we have seen how we can send elements to a function. On top of all that we have looked at how to return a value from a function. All this is fairly basic but not too out of reach that we can't do something with it.

And remember what I said above about using call and spawn…

"… use spawn to execute any function that has a sleep command inside of it, and if you want the function to return a value."

Global Functions

From here I will try and show you something useful and practical using global functions. For now think of Global functions as functions that are accessible from any location on the same machine. We are not going to talk about anything more for now, but I will expand on this in a later chapter.

So, Global functions….

Our current function '_hinting' is only accessible from our addaction when we call the function.sqf script. Even if we placed the _hinting function into its own personal script and then loaded it into memory when the game started, we wouldn't be able to access it. This is because it is local or private and can only be accessed from the same script. This is fine if you need to continually access a function within the same script, but what do we do if we require a function located elsewhere.

Remember the difference between local and global variables; one has the '_' whilst the other doesn't. This is the same for the function names. So create a new file called FunctionsOnLoad.sqf. In this file cut and paste across your _hinting function. Make sure the _hinting function no longer exists in the function.sqf file.

In your FunctionsOnLoad.sqf, where you should have now placed the _hinting function, amend the name of the function to this…

```
FNC_hinting = { etc…..};
```

Now save the FunctionsOnLoad.sqf file and go back to your Function.sqf. Now amend your code so that the calling of the _hinting function is now…

```
_total   = [1,2,3] call FNC_hinting;
_total1 = [1,2,3] call FNC_hinting;
_total2 = [1,2,3] call FNC_hinting;
_totalFinal = [_total, _total1, _total2] call FNC_hinting;
hint format ["Returned Value: %1", _totalFinal];
```

Save the function.sqf, and open up your init.sqf file. Inside type the following…

```
execVM "FunctionsOnLoad.sqf";
```

*Note – using execVM to preload functions will be replaced later on with a different command.

Save the init.sqf, and alt-tab back into the editor, click preview and once in game execute the function addaction again and see what happens… Yep Eighteen (18) is hinted on the screen once again. No I'm not putting a frigging screen shot of a hint showing 18!!!!

Breaking down the code…

Firstly making the _hinting function global - We added 'FNC' to the front of '_hinting', this instantly makes it global. We could have put anything in front of it, e.g 'FOCK_hinting' or 'WooHOO_hinting', it's just an identifier or 'Tag' so we can later identify to ourselves (and others) that this is a function. We could have changed the name altogether and called it 'myAwesomelyCrapFunc', all that matters is that it doesn't have '_' at the front of it, which changes it from local to global. This is exactly the same for variables. By making a variable global it becomes accessible from the same computer no matter where it is located, and as long as it has been loaded into memory.

Moving on…

We then changed the calling of the _hinting function to match its new name FNC_hinting. We did this in the function.sqf file.

```
_total   = [1,2,3] call FNC_hinting;
```

Here, we have changed the call to FNC_hinting. This is obviously our function which is located in FunctionsOnLoad.sqf, not in the same script. As the _hinting function no longer exists in this file, we cannot access that and as such if we did the computer would return an error. So, by changing the name to match that of our global function, it will now look into the computer's memory to find that function.

Loading 'FNC_hinting' into memory - We did this by executing the FunctionOnLoad.sqf from the init.sqf. By typing execVM "FNC_hinting.sqf", we have asked the computer to execute that file. When it does the FNC_hinting.sqf loads up and the computer sees that we have a function in there. The function on its own won't do anything; the computer loads that into memory and stores it, whilst making it available to other scripts on the same computer.

Therefore when we now start the game, the 'FNC_hinting' function is loaded into memory. When we use the addaction in game, the function.sqf file is executed, which in turn calls the 'FNC_hinting' from memory. And due to it being preloaded into memory it is generally quicker than a script being executed. The sum is returned each time from our 'FNC_hinting' function to our function.sqf, and a total is hinted on screen.

Hurrah a global function has been made! You can now call this many times and from different locations within your game mission folder.

Using what we have learnt so far…

Making a Patrolling Function

Finally something useful!!

Just so you know what to expect… we are going to make a patrolling function that will call another function which will return a random position within a circle. These returned coordinates will then allow our patrolling script to plot a position for our soldiers to move.

So, the two functions – one returns the coordinate and the other moves our AI Soldiers into those coordinates.

Let's start by making the Random position within a circle…

At this point I'm giving you a function that I have already made and use successfully in my A3 MP [COOP/TVT] Cache Hunt (http://www.armaholic.com/page.php?id=19875&highlight=FOCK) – feel free to download and test the mission and see how some of the script works. But please if you are using this function, I would appreciate that you would leave the name of the function ☺. It's the small things.

In any case, and the cheap excuse to self-publicise, type the following code inside of the FunctionsOnLoad.sqf…

```
FOCK_RndCirclePos = {
                        //randomise area of cache
                        //picks a random point within a circle
                        _objectsPos = _this select 0;
                        _mkrRadius = _this select 1;
                        _rndRadius = [-1,0,1];
                        _rnd = _rndRadius select (floor (random (count _rndRadius)));
                        _r = round (random _mkrRadius);
                        _Xp = _r * _rnd;
                        _Y1 = _r *_r;
                        _Y1A = _Xp *_Xp;
                        _Y1B = _Y1 - _Y1A;
                        _Y2 = sqrt _Y1B;
                        _Yp = _r - 2 * (_Y2);

                        //return position
                        _pos = [(_objectsPos select 0)+_Xp, (_objectsPos select 1)+_Yp, 0];
                        hint format ["position returned: %1",_pos];
                        _pos
                };
```

The two lines in red are as follows : - _objectsPos = the position around which you want your Soldiers to patrol, and _mkrRadius = the distance away from the object you want to patrol around. You will be passing these two elements to this function from your patrol script. You should know there are at least two elements being passed as you have _this select 0 and _this select 1, which we know indicates an array is being passed to this function.
I won't go on any more about this function; all you need to do to use it for yourself is to pass it an object's position [X,Y,Z] as the centre point, and then pass the distance you wish to have as the maximum radius away from the object you also passed to the function. It will then calculate a random position within that circle.

Onto creating the Patrolling script.

Next type the following after the 'FOCK_RndCirclePos' function.

```
FOCK_Patrol =
{
_unit = _this;
Hint "Patrolling";
_distance = [20,30,40];

            while {alive _unit} do
            {
             _countArray = count _distance;
             _rndSelectArray = _distance select (floor (random _countArray));
             _patrolArea = [getpos AmmoSups, _rndSelectArray] call Fock_RndCirclePos;
             (_unit) doMove (_patrolArea);
             _unit limitSpeed 3;
             sleep 8;
            };
};
```

Looking at the above there shouldn't be anything in there you don't understand if you have followed the tutorial thus far.

Breaking the code down…

We named this Global function 'FOCK_Patrol,' and then made it a function by placing the preceding code within brackets.
_unit = _this :- '_this' should tell you that something (singular) is being passed to the function. As there is no select command following '_this', it should indicate to you that only a single element/item is being passed, and not an array. Whatever is passed to the function we initialise _unit with that passed item. In this case it will be our AI soldiers, not as an array, but individually. Each soldier individually will get passed to this patrol function, so they can all randomly patrol around.

_distance = [20, 30, 40] :- this is clearly an array of three numbers. This will be our distance from the Ammobox that we will patrol around. The chosen distance will be passed to our 'FOCK_RndCirclePos' function as detailed above.

while {alive _unit} do:- You certainly should know that this is a while-loop, which means we are calling a certain piece of code continually until a condition is not met. In this case our individual AI soldiers that are passed to the function, when they die it will stop for them, not the other AI Soldiers – as each AI soldier is passed to this function independently of the others.

_countArray = count _distance :- count the number of elements in the _distance array – we know there are three (3) elements, those being the values of 20, 30, and 40.

_rndSelectArray = _distance select (floor (random _countArray)):-
This piece of code is slightly more complicated but if you break it down it makes sense. Working backwards through the separate brackets…

(random _countArray) :- This gives us a random float number between Zero (0) and three (3) (which is the count (3) returned from _countcoutArray).
Floor (returned value from random _countArray):- this returns an integer rounded down from the number obtained from (random _countArray). Since the highest number is three (3) it will always be rounded down to two (2) as the floor command rounds down numbers to nearest integer. So we therefore will never get three (3). We don't want three (3) as although the _distance array has three (3) elements its positions are [0,1,2]. Selecting three (3) would cause an error, because this means selecting the fourth element in the array.
_distance select (*) : - Now we are selecting a position from our array. We give the name of the array we want to select from "_distance"(Remember same as how we select using '_this'). We use the command select to select a position from within the _distance array. Then using the rest of the above code it will insert a random number between Zero (0) and Two (2), as the number always gets rounded down. Thus, it will chose either position 0, 1, or 2 randomly on each cycle of the while loop. We therefore get 20, 30, or 40 returned from this bit of code each time it cycles through the while loop.
_rndSelectArray: - the selected position from the '_distance' array is placed inside this newly created local variable. _rndSelectArray now holds that selected position, and when we use this it will refer to the position within that array. Remember as the computer loops through the while loop, this will continually get overridden.

_patrolArea = [getpos AmmoSups, _rndSelectArray] call Fock_RndCirclePos :- Well since we need to execute the 'FOCK_RndCirclePos' function and return a random position, we need to use the call command to call the function. Remember, when you want a function to return a value/position/array etc. you need to use the call command. We create the _patrolArea variable to hold the returned value, which in this case will be an array of coordinates.
[getpos AmmoSups, _rndSelectArray] :- getpos Ammosups is simply the coordinates of our ammo box, as passing the object on its own won't work. The 'FOCK_RndCirclePos' function needs coordinates/position rather than an object to calculate a random position. _rndSelectArray:- is clearly the selected position from within our distance array – which will be either 20, 30 or 40.
We have encased the position of the ammo box and the selected position in our _distance array so that they can both be passed to our 'FOCK_RndCirclePos' function. As an example this is what one cycle of the while loop could send…

```
_patrolArea = [[192.34,163,0], 20] call Fock_RndCirclePos
```

These two passed elements will now constitute the following in our 'FOCK_RndCirclePos' function…

```
FOCK_RndCirclePos = {
                        //randomise area of cache
                        //picks a random point within a circle
                        _objectsPos =  getpos AmmoSups;
                        _mkrRadius = _rndSelectArray;

                    ****othercode

                    };
```

OR if I changed it to visible values…..

```
FOCK_RndCirclePos = {
                        //randomise area of cache
                        //picks a random point within a circle
                        _objectsPos =  [[192.34,163,0];
                        _mkrRadius = 20;

                    ****othercode

                    };
```

(_unit) doMove (_patrolArea) :- At this point you should see that we have already passed (individually) our AI Soldiers to the 'FOCK_Patrol' function. Thus, we are therefore telling the computer to make each solider individually move to the returned coordinates from our 'FOCK_RndCirclePos' function.

_unit limitSpeed 3 : - Limitspeed (http://community.bistudio.com/wiki/limitSpeed) is a new command, that's why this line is in red. Simply, it limits the speed of an object (vehicle or unit). This stops our AI Soldiers running around to each patrol point. By making them slower it looks a little better.

Sleep 8 :- well this just pauses the script for 8 seconds before the while loop goes round again and calls another random position from the ammo box.

We then close off the while loop, and then close off the 'FOCK_Patrol' function.

*Note as this function ('FOCK_Patrol') has a sleep command in it we will have to execute it using spawn – NOT call., otherwise it will just ignore the sleep command. Furthermore, even if it the sleep command was not used, we would still have to use the spawn command. This is because when the first AI soldier is passed to the script, he enters the while loop, continually getting a new position and moving to that position until he is dead. The call command demands that we have to wait until that function finishes for him specifically before the next AI Soldier can be passed to his 'FOCK_Patrol' function. Remember each AI is technically running their own 'FOCK_Patrol' function independently of the other units. It's like each AI Soldiers is calling their own addaction for themselves, each calling a function that they are running specific to them.

If you did use call, all you would see is one (1) AI soldier running around patrolling. The rest of the Soldiers would be waiting for his code to finish. You could simply shoot that one (1) soldier and then the next will start patrolling, but obviously this is of no use to use. So SPAWN command it is.

Hopefully you can see the difference between call and spawn in these functions. You may even be wondering how we can use spawn to execute the 'FOCK_Patrol' function, and from inside the 'FOCK_Patrol' function use call to execute 'FOCK_RndCirclePos'. Don't we have to wait for that to finish? Yes we do, but the 'FOCK_RndCirclePos' is not in a loop, it finishes each time it is called, and returns a value /array that is used in the 'FOCK_Patrol' function Therefore we can use call in this instance.

Initial execution of the FOCK_Patrol function…

In our Array.sqf, which we call from an addaction, amend it so it looks like this…

```
_aiDudes = [Dave, Eric, Paul, Tom, Dick, Harry];
_newAiDudes = [];
for "_i" from 0 to 5  do
{
	_total = (count _aiDudes) - 1;
	_rnd = floor (random _total);
	_pick = _aiDudes select _rnd;
	hint format ["Soldier Picked: %1",_pick];
	sleep 2;
	(_pick) doMove (position AmmoSups);
	_aiDudes = _aiDudes - [_pick];
	_newAiDudes =  _newAiDudes + [_pick];
};

{
  waituntil {(((position _x) distance (position AmmoSups)) < 8)};
  _x spawn FOCK_Patrol;
}foreach _newAiDudes;
```

All I have done is removed all but one sleep command. The one left assists with slowing down the rate at which the units move to the ammo box; otherwise the code gets executed so fast all AI Soldiers will move all together. Everything else remains the same except for the last but one line.

```
_x spawn FOCK_Patrol;
```

You should see from this line that we execute our 'FOCK_Patrol' Function via the Spawn command as discussed above. '_x' – remember this stands for each one of our AI Soldiers in our newly created _newAiDudes array. And using forEach will loop through that array, in turn independently replacing '_x' with one of our selected Soldiers, which then executes the function 'FOCK_Patrol' for that specific AI Soldier.

It's as if we simply typed…

```
Dave spawn FOCK_Patrol;
Eric spawn FOCK_Patrol;
Paul spawn FOCK_Patrol;
Tom spawn FOCK_Patrol;
Dick spawn FOCK_Patrol;
Harry spawn FOCK_Patrol;
```

You should see that the computer is telling each AI Soldiers to spawn the function for themselves, each getting different random distances and positions from the ammo box to patrol. Save the Array.sqf, and save all other scripts. Remember the init.sqf is executing the FunctionsOnLoad.sqf when the game starts, which is compiling/storing all our functions in that file.

Alt-Tab back into the game editor, click preview and then run execute our 'Rnd Array Select' addaction – not the 'function' addaction, which is our hinting function. Watch what happens…

Yep, they are slowly (limitspeed) walking around and waiting, and then moving to another location.

You have successfully called an addaction that gets your AI soldiers to move to an ammo box. Once there they will all independently go about their own patrolling duties.

Chapter 2 Conclusion…

You have now learnt about the methods to Call functions, Spawn functions, return values/arrays/objects from functions, and pass elements to functions. You have used the knowledge you have learnt surrounding Arrays to create an array by adding and subtracting from/to other arrays, and selecting elements from and array. You have learnt about the forEach command and how to loop through arrays easily selecting the necessary elements. More so you have witnessed the quick and efficient fashion in which to do it.

Thanks Mike J [FOCK] - http://fockers.moonfruit.com/