Coding Basics

By austinXmedic

Table of Contents

Intro

Alright, you are probably here reading this because you want to learn how to make some code for missions in Arma 3.

I will try my best to explain things and give sample code for you to experiment with.

Also I use some syntax highlighting: Red is a command/statement of some sort, and purple represents variables.

NOTE: This guide probably won't contain everything under the sun from the first release. Most of this was just written off the top of my head without lots and lots of planning first. If you think anything is missing from this then go ahead and post down in the comments.

Helpful Websites/ Where to Find Code

Learning SQF the best way is by doing, so disassembling/tearing apart scripts that are already made then mashing a few together to make a new script is a good way to learn it. There is Tons of sample code on the web

armaholic.com (which I would assume you already went to since that's where I released it) is one good site to use for finding anything Arma related really. It has scripts that you can go trawling through to learn.

The BI Community Wiki (https://community.bistudio.com) is also a really good place to go when you are not sure what a particular command does, though the documentation can be a bit spotty at times.

The BI Forums and Armaholic forums are also really good places to go to get help with scripting.

When posting to them you should try to describe your problem as much as possible and tell exactly what you are trying to achieve.

There is even some programs people have created to automate the tedious task of creating the init.sqf and Descripton.ext files, as well as helping you to build the rest of your mission, then allowing you to add onto what it creates with your own scripts.

There are a few other sites where some people post scripts and stuff for others to use

SQF File Structure

Like any programming engine, the game has some rules for its structure.

Curled braces ({ }) are able to group code into blocks.
Statements are followed by semicolons(;)
Semi-colons also signal the end of a line of code.

Example:

```
 STATEMENT 1;
 STATEMENT 2;

BLOCK (if statement perhaps)
{
 STATEMENT 3;
 STATEMENT 4;
};
```

Since the if-statement is itself a statement, you can also create nested if-statements (If statements inside If-statements).
Example:

```
if (alive player) then
{
 if (speed player > 0) then
 {
 hint "The player is alive and moving!";
 }
 else
 {
  hint "The player is not moving!";
 };
}
else
{
  hint "The player is dead!";
};
```

SQF File Structure Continued

In some cases you may want to check a variable for several values and execute different code depending on the value. With the above knowledge you could just write a set of if-statements.

```
  if (_color == "blue") then
  {
  hint "What a nice color";
  }
  else
  {
  if (_color == "red") then
  {
  hint "Don't you get aggressive?";
   }
};
```

The more values you want to compare, the longer this block of code becomes, and the longer it takes to execute, and the more impact on performance it has. That is why the simplified switch-statement was introduced.

The switch-statement compares a variable against different values:

```
switch (VARIABLE) do
{
  case VALUE1:
  {
  STATEMENT;
  ...more code
  };

  case VALUE2:
  {
  STATEMENT;
  ...more code
  };

  ...continued
};
```

You may also want a default value if what was in the switch variable doesn't match anything that was given, this is pretty simple: you just add

```
  default
  {
```

```
  STATEMENT;
  ..
```

SQF File Structure Continued

```
};
```

to the bottom of the sequence above.
so now it would look like this:

```
switch (VARIABLE) do
{
  case VALUE1:
  {
  STATEMENT;
  ...more code
  };

  case VALUE2:
  {
  STATEMENT;
  ...more code
  };

  ...continued
  default
  {
  STATEMENT;
  };
};
```

Loops are used to execute the same code block for a specific or unspecific number of times.
while-Loop
This loop repeats the same code block as long as a given condition is true.
NOTE: ALWAYS USE SLEEP OR WAITUNTIL COMMAND SOMEWHERE IN THE LOOP, OTHERWISE IT WILL BASICALLY CRASH THE GAME

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

If the CONDITION is false from when the game gets to the while-loop part, the statements within the block of the loop will never be executed.

Because the test of the while expression takes place before each execution of the loop, a while loop executes zero or more times. This differs from the for loop, which executes one or more times.

SQF File Structure

Example:

```
_counter = 0;

while {_counter < 10} do
{
  _counter = _counter + 1;
  hint format ["Count: %1",_counter];
  sleep 1;
};
```

That brings us to the next part of SQF File Structure, the for-loop

The for-loop repeats the same code block for a specific number of times.

This is a complex one, there is optional parts of this one, so if you want it simple, don't use the one below.

```
for [{BEGIN}, {CONDITION}, {STEP}] do
{
  STATEMENT;
  ...
};
```

SQF File Structure Continued
BEGIN is a number of statements executed before the loop starts
CONDITION is a condition evaluated before each loop
STEP is a number of statements executed after each loop

```
// a loop repeating 10 times
for [{_i=0}, {_i<10}, {_i=_i+1}] do
{
  Player  globalChat format["%1 for-loop",_i];
  sleep 1;
};
```

SQF File Structure End

SIMPLE for-loop:

```
for "VARNAME" from STARTVALUE to ENDVALUE do
{
  STATEMENT;
  ...
};
```

Data Types

I'll explain some data types you'll use when coding for the game.

These Datatypes will be mainly inside variables to hold data for commands.

OBJECT - This can be anything from a water bottle to a tank to a player (sometimes referenced through the name that was given in the parameters box that pops up in the editor).

ARRAY - Basically this is a list of data elements (can be any data type), and they can store different data types in different elements (elements are the things in the array).

Arrays can also have arrays inside them (nested arrays), these can cause some coding issues, especially with people who aren't familiar with them.

STRING - Obviously, it's a set of plain text contained in quotes

NUMBER/SCALAR - Obvious, it's a number

These data types can be destroyed by doing (recommended once the data variable is no longer needed for any other part of your mission)

```
MyVariableName = nil;
```

Files

Arma 3 uses SQF file format for scripts. You can affiliate the filetype with notepad, and it will work fine, though I recommend notepad++ as it has a SQF plugin that can catch basic mistakes and features highlighting of key parts of the code.

Some files you'll probably be using often will be

Descripton.ext

https://community.bistudio.com/wiki/description.ext (runs before the mission begins, errors will cause a game crash)

Other event scripts

https://community.bistudio.com/wiki/Event_Scripts

init.sqf (runs during the briefing before the mission begins, this is probably the most important of the batch as you'll use this to execute all other scripts you need ran at mission start)

There are more than that, and those can be found at Bohemia's wiki, but they will probably be used far less than init.sqf

Any errors in the code will stop execution of code beyond where the error happened inside the script where the error occurred. (The hint command can be helpful here if an error message with the black box doesn't pop up for some reason).

Variables

Variables are things that hold some data inside memory. These are used pretty frequently depending on what you are doing.

Variables support text and numbers, and possibly support other weird symbols, but I wouldn't recommend using the symbols.

Using an underscore before the text part of the variable where you name it will make it local to the scripts it is in.

This means that other scripts won't be able to reference this variable, as for them it doesn't exist.

Obviously no underscore makes it global, and all scripts will be able to reference it.

I recommend you make it local to the script it's in unless it's required for another script to be able to reference a variable.

Example:

```
myVariableName = MyVariableValue;
```

Passing Variables to a Script

You'll probably need to pass some variables to a script pretty often as most scripts need a variable to reference to run code on.

In most coding/programming languages your values (or indexes for correct terminology) in arrays do NOT start at 1, they start at 0 and count up. (Indexes inside an array start at 0 and count up by 1 for each thing there is inside it)

When passing variables to a script, the array containing the variables is referenced as _this. You'll want to be able to reference this data easier later, so you'll transfer the data from the variable in the _this array to another variable inside the script.

So here's an example of how to retrieve data from this array:

```
_VARIABLE = _this select 0; //-- Select the first thing that was in the array of elements passed
```

Comments

You probably noticed how I put

// behind my lines of code in the section above. That is a comment

A comment is any free text which is ignored by the game. In SQF you can write comments using the command comment.

Comments can be used to, well, comment the code, put notes in, or something like that.

Example:

comment "This is a comment";

If a script is loaded with preprocessFile, execVM or spawn, you may also define some other comments:

Line comments

A line comment starts with // and makes the rest of the line a comment, it doesn't need a semi-colon at the end.

Block comments

A block comment starts with /* and ends with */. All text in between is considered a comment.

Examples:

// This is a line comment

/*

This is a

block comment

*/

Functions

A function is something defined in code which carries out an instruction or set of instructions and may or may not have inputs and outputs. The built-in functions which are provided by Arma 3 are referred to as commands, and are called in a different manor to functions which are defined within scripts.

Functions main purposes is to really most of the time reuse code, saving in file size and performance by not having duplicate code all over the place. It is also nice to have functions to do some code, as if the function is updated, it's updated for all scripts, and doesn't need to be updated for each individual script.

Functions as files are functions stored within a file. These are usually used for larger and more complex functions. The code is evaluated in the same way, however, there are additional commands which must include the file before the function itself can be called (loadFile, preprocessFile, etc).

Eg.

```
MyScriptVar = preprocessFileLineNumbers "myscript.sqf";
//MyFile.sqf

// Code here...
// moar code!
```

Inline functions are functions are technically code which is often stored within a variable or declared as a function parameter. Inline functions operate the same way as functions-as-files as both are evaluated in the same way, but the difference is that inline functions are stored within parentheses{}, and can also have multiple functions stored in the same .sqf file, whereas functions-as-files do not require the parentheses, and cannot have more than one function in the same file.

```
MyVariableName =
{
  _parameterOne = _this select 0;
  hint format [" Param 1 value: %1",_parameterOne];
  // Code here..
  //moar code!
  // moar code!
};
```

And of course, if the function doesn't need any parameters, you can just leave the brackets blank

```
[] call MyVariableName;
```

Error Box

In Arma scripting can get a bit complex. If there's a little mistake somewhere it will break the whole script until it's fixed. Luckily you don't have to go trawling through your code to try to guess where the mistake is.

The error box is a useful tool for finding errors in the code. It is turned off by default, so to turn it on you use the launch parameters -showscripterrors.

The path to the script where the error happened is usually the first thing inside the box

The error is usually located on the second line of the error box, this also states what line the error was and the game will attempt to tell you a reason for the error, but most of the time it will default to "missing ;" instead of something more logical. This error message usually means that there was a typo somewhere in the code when a command was referenced, though not always.

Clean Code

Clean code refers to code that is formatted properly, so it is easily readable.

A piece of dirty code would look something like this:

```
{if(_x == player) then {
code1; code2; code 3;
};} foreach allUnits;
```

Take that and paste it about 30 times and see if its easily readable, its not.

It'll be very difficult to find issues as well, as it is messy and hard to read, and the game's error box will only state the line the error is on, so if all the code is clumped together on one line, it will only say that the error is on that line, rendering it basically useless as you'll be scrolling the whole thing trying to find the issue.

Clean code would look something like this:

```
{
  if(_x == player) then
  {
    code1;
    code2;
    code3;
  };
} foreach allUnits;
```

A bit easier to read, yes? This will also make it easier to debug later as the error box will be able to effectively tell you where there's a problem.

It should also be noted that it's much easier to start with something simple that does what you want (then build upon it), it can always be formatted later, but it can also be a bit easier if it's done as you go. Which one you prefer to do is your choice really.

Some More Complex Scripting – Cinimatics

NOTE: I would only move onto this once you can start using scripts to manipulate objects in the game world and have some basic knowledge of SQF scripting.

Some Missions probably use cinematic scenes. You might think this is a hard thing to do, and it can be to some extent. Getting units/objects into the right positions and getting them to do what you want exactly can be a bit difficult and time consuming.

Cinimatic Scripting Commands

_varHere means that you can put whatever you like, but this will be how you will control the camera using other commands.

titlecut ["","BLACK OUT",fadeTime]; - this will fade the game so the player cannot see you spawning stuff in for the cutscene.

_varHere = "camera" camcreate POSITION; - kind of obvious, spawns a new camera for you to use.

_varHere cameraeffect ["internal", "back"]; - this controls some of the effects for the camera, such as using NV or IR. The one above is just normal view without any other effects.

showcinemaBorder false; - If you want to show black lines on the bottom and top of the screen, change false to true.

_varHere camPrepareTarget POSITION; - Sets a target for your camera to point at, if the camera is moving, it will rotate around to keep this point centered.

_varHere camPreparePos POSITION - Sets the Camera's actual position

_varHere camPrepareFOV 0.740 - I'm pretty sure this controls the focus/zoom of the camera, 0.740 is pretty good for most situations.

_varHere = camCommitPrepared 5; - This will be how fast it will take for the camera to move to the camPreparePos position, this is how you will control how fast the camera moves around to new positions.

_varHere cameraeffect ["terminate", "back"]; - Use this to start destorying the camera once the cutscene is done

camDestory _varHere - use this to finish destorying the camera.

NOTE: If terminating the camera, you may want to use titleCut["","BLACK OUT",fadeTime]; to fade the camera out before destorying it. To get rid of the effect just change BLACK OUT to BLACK IN.

Cinimatic Scripting – Helpful Code

You will need to rip positions out of the game world to use with your cutscene, for this I made a function that when used from the action menu will copy it to your clipboard.

```
AUSMD_fnc_copyPos =
{
  _unit = [(getposASL player) select 0,(getposASL player) select 1,0];
  copyToClipboard format["%1",_unit];
  hint "Position copied";
};
```

When adding units to the cutscene, you'll probably need to get the right direction, for that use the function below.

```
AUSMD_fnc_copyDir =
{
  _unit = getDir player;
  copyToClipboard format["%1",_unit];
  hint "Direction Copied";
};

player addAction ["Get pos",{[] call AUSMD_fnc_copyPos}];

player addAction ["Get dir",{[] call AUSMD_fnc_copyDir}];
```

Cinimatic Scripting – Helpful Code Continued

Here is a sample script:

```
titlecut ["","black out",2];

sleep 2;

_camera = "camera" camcreate [3380.36,13238.9,0];
_camera cameraeffect ["internal", "back"];

showcinemaBorder false;

_actor1 = "B_Soldier_F" createVehicle [3379.46,13236.8,0];
_actor1 setDir 238;
_actor2 = "B_Soldier_F" createVehicle [3377.98,13237.7,0];
_actor2 setDir 181;
_vehicle = "I_Truck_02_covered_F" createvehicle [3375.28,13230.3,0];
_vehicle setDir 236;

_camera camPrepareTarget _actor2;
_camera camPreparePos [3380.36,13238.9,1];
_camera camPrepareFOV 0.740;
_camera camCommitPrepared 0;

titlecut["","BLACK IN",2];

_actor2 playMoveNow "Acts_WalkingChecking";

_camera camPrepareTarget [3373.79,13231,1];
_camera camPreparePos [3372.67,13233.4,1.5];
_camera camPrepareFOV 0.740;
_camera camCommitPrepared 14;
sleep 14;
_camera camPrepareTarget _actor2;
_camera camPreparePos [3372.67,13233.4,2];
_camera camPrepareFOV 0.740;
_camera camCommitPrepared 10;
sleep 10;titlecut["","BLACK OUT",2];

sleep 2;
```

Power Failure Scripting

Ever wanted to plunge an area into darkness after your team destroys a transformer nearby? With this basic script you can

```
_centerPos = _this select 0;

blowlights =
[
	"Lamps_base_F",
	"Land_PowerPoleWooden_L_F",
	"PowerLines_base_F",
	"PowerLines_Small_base_F"
];

_nearestGenerator = nearestObject [_centerPos,"Land_spp_transformer_F"];

_allLights = nearestObjects [_nearestGenerator, blowlights, 700];

waitUntil{damage _nearestGenerator >= 1};

hint "Generator Down";


{
	_x setHit ["light_1_hitpoint", 0.97];
	_x setHit ["light_2_hitpoint", 0.97];
	_x setHit ["light_3_hitpoint", 0.97];
	_x setHit ["light_4_hitpoint", 0.97];
} foreach nearestObjects [_nearestGenerator, blowlights, 1000];
```

The script works by setting up an array of lights to blow out after the power goes down. The script also searches out from the position passed to it via execVM to find the nearest solar power transformer to that position. When the generator is destroyed (damage = to 1), all lights within 700 meters will go out by setting the light hit points to 0.97 to break them partially.

Disco Lights!

In this section we will be going over how to make your own disco lights.

We'll be using some pointlights on the ground to do what they do best: emit light out in all directions

Create a script called popflare.sqf inside your mission file.

Put this code inside it:

```
_position = _this select 0;
_mybrightness = _this select 1;
_mybrightnessrandom = _this select 2;
_randomString = format ["%1%2",random 100, random 100]; //get random var, 10000 combinations
_randomString = "#lightpoint" createVehicle _position;
_randomString setLightBrightness _myBrightness;
_randomString setLightAmbient[random 1,random 1,random 1];
_randomString setLightColor [random 1,0.75,0.25];

while {alive _randomString} do
{
	_random = _myBrightness + (random _mybrightnessRandom) - (random
_mybrightnessRandom);
	_randomString setLightBrightness _random;
	_randomString setLightAmbient[random 1,random 1,random 1];
	_randomString setLightColor [random 1,0.75,0.25];
	sleep 0.5;
};
```

This script will take 3 input values, a position, a brightness value, and a random brightness value. As you can see it randomizes this random value and adds it then takes away another random value from that random value you selected, this will get a good amount of randomization in. It then sets the light brightness, but also sets the light ambient and light color values, but those are done using the random command directly, and cannot be edited by the user easily (although you can change them in the physical script to whatever you want.

The loop will repeat while the light is alive or otherwise until it disappears and is deleted

Inside init.sqf write:

```
nul = [getpos player,1,0.5] execVM "popflare.sqf";
```

That will spawn one light at your feet on mission start, and it should start changing colors and using different brightness's over time. You can paste that code and change the second and third values as

Disco Lights Continued!

much as you like into the debug box and keep creating more of these lights as the script creates a random value for every single one of the lights,

This is really useful when you don't need to reference that specific light ever again, but need to create multiple lights using only one script.

This isn't always a good option though, in most cases you will not use this method to get variables to use, but in this case it's fine.

As an alternative you could probably create an array of lights spawned then delete them all once no longer needed if you wanted to use this method but still be able to remove the lights.

Credits/Thanks

A big thank you to the many people on the BI forums and Armaholic forums who have helped me over the years with getting better at scripting and helping me to solve problems with scripts.

A thank you to Bohemia and the community for putting together some good documentation on the biki, it has definitely been a huge help to me with scripting.