

Arma 2: SQF scripting guide. v.3.2

INTRODUCTION TO THIS DOCUMENT	2
Conventions used	2
SCRIPTING	3
What you need.....	3
Mission location	3
Common topics for scripting	4
Examples	5
My first Procedure	5
My first function	6
My first trigger condition using a Function.....	7
My first combined Function and Procedure.	10
Procedures and Functions explained	11
Procedures.....	11
Functions	12
Variables	13
More on variables.....	14
Strings.....	14
Arrays	15
Numbers.....	16
Objects.....	16
Magic variables	17
Advanced usage of variables	17
Even more information about variables	17
Control structures	17
if-statement	17
if-else statement.....	18
else-if statement.....	18
switch-statement.....	18
for-loop	19
forEach-loop	19
Even more on control structures	19
Operators.....	20
Logical Operators	20
Comparison Operators	20
Even more on Operators	20
When Do I Need Scripting?.....	21

Best practices.....	22
Naming conventions	24
APPENDIX.....	25
Related wiki articles used in this document	25
Control structures.....	25
Data types.....	25
SQF Syntax	25
Operators.....	25
Example of Crew command wiki article.....	25
ABOUT THE AUTHOR.....	26
VERSIONING.....	27
To-do:.....	27

Introduction to this document

This guide aims at people without or little scripting experience. You don't need to have any previous coding experience with say; java-script, vb-script. However, if you have worked with SQF previously you might still find this guide interesting.

The guide also makes references to articles on the [BIS:Wiki](#), where you should look to find more information about commands, and to search for commands which may cover your needs. The scripting commands are quite easy to understand, search and you will most certainly find what you are looking for.

Please read the guide thoroughly, when finished you should be able to make your first more advanced missions which includes both procedures and functions, and have more understanding about the data types and structure of the code.

All examples found in this guide are working and may be used as you want.

Conventions used

BIS commands are in *Italic* e.g.

format

Both commands and code is in a different font and size and color where applicable.

e.g.

```
_myString = "";
```

Scripting

What you need

- Notepad or a text editor of your choice.
- A bookmark to [Scripting commands ArmA 2](#)
- “Hide extensions for known file types” turned off in windows.
Windows explorer > Tools > Folder Options
View tab
- Add -showScriptErrors to your ArmA 2 shortcut.
e.g. “...Bohemia Interactive\ArmA 2\ArmA 2.exe” -showScriptErrors
- Basic knowledge about the mission editor, read more in the [ArmA: Mission editor](#) –article.
- To know what you want to accomplish with your script.
It’s better to lay out a development plan for your script before starting instead of doing it “on the fly”
- Patience

Mission location

Windows Vista

%userprofile%\Documents\ArmA 2 <profile>\<player>\missions\<mission name>.<island>

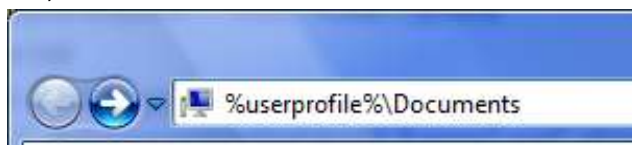
Windows XP

%appdata%\Documents\ArmA 2 <profile>\<player>\missions\<mission name>.<island>

e.g.

..\Documents\ArmA 2 Other Profiles\Taurus \missions\ MyFirstSQFTest.utes

If you are unsure, open a windows explorer and copy the line corresponding to your OS in the URL box, as such.



Common topics for scripting

There are two types of scripts in ArMA 2.

- Functions
- Procedures

These two types will be explained in detail later in this guide.

The functions and procedures are compiled by

```
compile preProcessfile "filename.sqf"  
compile preprocessFileLineNumbers "filename.sqf"
```

preprocessFileLineNumbers will return the failing line if a compile error, or misuse of a command is found in the script.

Or in the case of when you used *execVM* you don't need to explicitly compile the files.

This is done by the game engine with the *preprocessFileLineNumbers* command.

Input arguments are fetched by the "magic" *_this*

e.g.

```
_this select 0;
```

Which will fetch the first element of the arguments array sent into the function or procedure.

Arguments are always sent as an array, you will see this later on as you progress through this guide.

If you chose to put the script files in folders, they are then accessed by relative path of the "mission.sqm"-file

e.g.

```
MyFunction = compile preProcessfile "MyFunctions\MyFunction.sqf";
```

Files compiled with the above mentioned compile options:

preProcessfile, *preprocessFileLineNumbers*
can include comments(like the ones from C++,Java,and Javascript)

e.g.

```
//Single line comment
```

```
/*Comment covering  
one, two or  
even more lines */
```

Use comments to document your script, what it does and why, which arguments it requires, even author information and etc can be used.

All scripts will be treated as complete of the script engine after the last line has been reached.

Use of variables, please see section about Variables.

Examples

These examples might look overwhelming at a glance, but follow the steps and you'll see that it works. Details about what the code does and why will be covered later.

The scripts in the examples are narrowed down because they should be easy to read, feel free to see the comments in the source-files found in the .zip-file.

My first Procedure

Start Arma 2.

Open up a new mission in the editor, Utes preferably.

Place one unit of a side of your choice and put in its init line:

```
MyProcedureHandler = [this] execVM "MyProcedure.sqf"
```



Press "OK"

Place an empty Hmmwv next to him, and save the mission as "MyFirstSQFTest"

(alt+tab) to return to the desktop, and browse to your mission folder described in "Mission location"

Open the folder called "MyFirstSQFTest.utes"

In it you will have one file called "mission.sqm"

Create a new text file, name that "MyProcedure.sqf"

Then right-click and open the file, chose "notepad" as the program to open this file with.

In this file write.

```
_myGuy = _this select 0;  
//This is a while loop which will run until _myGuy dies  
while {alive _myGuy} do {  
    //HintSilent will show the text in the upper right corner  
    hintSilent format ["Current speed: %1", round(speed (vehicle  
_myGuy))];  
    //And updates the value every second.  
    sleep 1;  
};
```

Save the file (ctrl+s) and close it.

Now (alt+tab) back to ArMA 2 and the editor and preview the mission, run around a bit and enter the car you placed and see the current speed(km/h) in the upper right corner.

You will notice the speed to be the same as the car UI but just updated a little bit slower, you can lower the `sleep 1;` to `sleep 0.125;` to make it update the value more often.

Exit the mission.

My first function

Now place a Radio Alpha trigger in your mission.

In the “On Act” field, write:

```
player sideChat format ["%1",[player] call MyFunction]
```



Click “OK” and save the mission.

(alt+tab) to return to the desktop, and go back to your mission folder, create a new file called “init.sqf”

This is the init file of your mission, and will run after the unit init lines have been processed.

Open "init.sqf" as you did with the other file with Notepad.

In this file write.

```
MyFunction = compile preProcessfile "MyFunction.sqf";
```

Save the file (ctrl+s) and close it.

Create another file, call this "MyFunction.sqf"

Open it as you did with the other file, and write in it.

```
_myGuy = _this select 0;

//Variable to hold the out value
_myOutStr = "I'm not in a vehicle!";
//Checks whether or not the unit is in a vehicle
if(_myGuy != (vehicle _myGuy)) then{
    _myOutStr = "I'm in a vehicle!";
};
//Return value
_myOutStr
```

Save the file (ctrl+s) and close it.

(alt+tab) back to the editor and preview the mission.

Use Radio Alpha and see what you say.

Then enter the car and use Radio Alpha again.

You will promptly say if you are in a vehicle or not, to your current side.

Exit the mission.

My first trigger condition using a Function

One way of making good use of functions is to let them handle the condition in triggers.

The editor triggers can be set to trigger when a BLUFOR unit enters it, but what if you want to check more things like:

- There need to be at least 2 BLUEFOR units in the trigger area
- The player should use the radio to report he is done with his duties.

The code statement could be written directly in the condition box, but for more complex conditions it is easier to let a function do the job.

Update your “MyFirstSQFTest”-mission by adding another soldier of the same side as you.



The "Insert unit" dialog box is shown with the following settings:

- Side: **BLUFOR**
- Faction: **USMC**
- Class: **Men**
- Control: **Non Playable**
- Info age: **Unknown**
- Vehicle lock: **Default**
- Rank: **Private**
- Unit: **Rifleman**
- Special: **In Formation**
- Name: (empty)
- Skill: (empty)
- Initialization: (empty)
- Description: (empty)
- Health/armor: (empty)
- Fuel: (empty)
- Ammunition: (empty)
- Probability of presence: (empty)
- Condition of presence: **true**
- Placement radius: **0**

Buttons: **OK**, **Cancel**

Click “OK”.

Now add the condition trigger.

Place this trigger somewhere in the middle of the hangar.

In the condition field write:

```
([thisList,2] call MyConditionFunction)
```



The "Edit trigger" dialog box is shown with the following settings:

- Axis a: **20**
- Angle: **0**
- Axis b: **20**
- Activation: **BLUFOR**
- Once: **Once**
- Repeatedly: (unchecked)
- Present: (checked)
- Not present: (unchecked)
- Detected by **BLUFOR**
- Detected by **OPFOR**
- Detected by **Civilians**
- Detected by **Independent**
- Detected by **Timeout**
- Min: **0**
- Mid: **0**
- Max: **0**
- Type: **End #1**
- Name: (empty)
- Text: (empty)
- Condition: **[[thisList,2] call MyConditionFunction)**
- On Act: (empty)
- On Dea: (empty)

Buttons: **Effects**, **OK**, **Cancel**

Please note that you should not place “;” in any condition fields.

Click “OK”.

Add another radio trigger.

Make this one Radio Bravo, and in its OnAct write:

ImHere = **true**;



Click "Ok".

Your mission should look something like this in the editor.



Save the mission.

It is now time to do some more scripting.

(alt+tab) to return to the desktop.

First thing you need to do is to update the "init.sqf" file.

Open this file as you did with the others, and add:

```
MyConditionFunction = compile preprocessfile "MyConditionFunction.sqf";  
ImHere = false;
```

Save the file (ctrl+s) and close it.

Then create a new sqf-file, call this "MyConditionFunction.sqf", write in it.

```
_triggerList = _this select 0;  
_condNumbOfUnits = _this select 1;  
  
_returnValue = false;  
_unitCount = (count _triggerList);  
//Checks if condition number of units equals the current count of units in  
this trigger  
if((_condNumbOfUnits == _unitCount) && ImHere) then{  
    _returnValue = true;  
}else{  
    if(ImHere) then{  
        player sideChat format ["Condition not met! Only %1  
unit(s) in the trigger!", _unitCount];  
        //Reset ImHere value  
        ImHere = false;  
    };  
};  
//Return  
_returnValue
```

Save the file (ctrl+s) and close it.

(alt+tab) back to Arma 2 and the editor, preview the mission.

Order your soldier to enter the hangar and use Radio Bravo.

The condition is not met and will not make the mission end.

Now join him in the hangar and press Radio Bravo again.

My first combined Function and Procedure.

Now when you have learnt how to code two functions and a procedure, it's time to put this knowledge together.

Instead of having to use the radio to see if you are in a vehicle, you want to see what type of vehicle you are in, as well as see your current speed.

(alt+tab) to return to the desktop.

Open the "MyFirstSQFTest"-mission folder again if not already opened.

For safety, copy “MyProcedure.sqf” and then open the file (not the copy), and change it as follows.

```
_myGuy = _this select 0;
//This is a while loop which will run until _myGuy dies
while {alive _myGuy} do {
    //HintSilent will show the text in the upper left corner
    _unitSpeed = round(speed (vehicle _myGuy)); //Added
    _vehicleType = (typeof (vehicle _myGuy)); //Added
    _inVehicle = [_myGuy] call MyFunction; //Added
    hintSilent format ["Speed: %1\nType: %2\nIn vehicle:
%3", _unitSpeed, _vehicleType, _inVehicle]; //Changed
    sleep 1; //And update the value every second.
};
```

Save (ctrl+s) and close the file.

Now make a back-up of the “MyFunction.sqf”, and change it like this.

```
_myGuy = _this select 0;
//Variable to hold the out value
_myOutStr = "No!"; //Changed
//Checks whether or not the unit is in a vehicle
if(_myGuy != (vehicle _myGuy)) then{
    _myOutStr = "Yes"; //Changed
};
//Return value
_myOutStr
```

Save (ctrl+s) and close the file.

(alt+tab) back to the editor and preview the mission, you’ll see your current speed, what kind of vehicle you are in, if not a vehicle, the type of soldier you are will be displayed. And if in a vehicle it will say “Yes”.

Enter the car and see the values change.

Exit the mission, or walk into the hangar and use Radio Bravo.

Procedures and Functions explained

Procedures

Executed from by using *spawn* or *execVM*, in order of game execution

Unit init lines:

```
MyProcedureHandler = [this] execVM "MyProcedure.sqf";
```

Procedures cannot be spawned from unit init lines unless the procedure is initialized before the spawn.

This will not be covered in this guide as of yet.

init.sqf:

```
MyProcedureHandler = [this] execVM "MyProcedure.sqf";
_myProcedureHandle = [MyVariable] spawn MyProcedure;
```

Triggers onAct ,onDeAct:

```
MyProcedureHandle = [MyVariable] spawn MyProcedure;
```

All procedures need to have a script handle, as shown above.

These script handles does not need to be unique unless you are going to use them in a later stage, as described below.

The script handle can be used to check if a script has completed, as follows

```
_myProcedureHandle = [MyVariable] spawn MyProcedure;  
waitUntil{scriptDone _myProcedureHandle};
```

Or the script handle can be used to terminate the script.

For example, if you want to terminate a script after 10 seconds, regardless if it's finished or not.

```
_myProcedureHandle = [MyVariable] spawn MyProcedure;  
sleep 10;  
terminate _MyProcedureHandle;
```

All code lines needs to be closed with “;”

e.g.

```
hint "Hello";
```

Procedures can be halted with

```
sleep X;
```

Where X is the number of seconds to halt(pause/sleep) the script, e.g.

```
sleep 10;
```

Procedures will run alongside the starting script, and for each “spawn” or “execVM” of them there will be a new instance of the script.

For example:

```
_myProcedureHandler = [_myGuy1] spawn MyProcedure;  
_myProcedureHandler = [_myGuy2] spawn MyProcedure;  
_myProcedureHandler = [_myGuy3] execVM "MyProcedure.sqf";  
_myProcedureHandler = [_myGuy4] execVM "MyProcedure.sqf";
```

The above example will spawn four instances of the MyProcedure script.

Functions

Executed by using call, in order of game execution

Unit init lines:

Functions cannot be called from unit init lines unless the function is initialized before the call.

This will not be covered in this guide as of yet.

init.sqf:

```
MyFunction = compile preProcessfile "MyFunction.sqf";
```

```
_myFunctionRetVal = [MyVariable] call MyFunction;
```

or

```
[MyVariable] call MyFunction;
```

Triggers condition:

Functions used here must return a value. e.g.

```
(([MyVariable] call MyFunction) == 0)
```

Triggers onAct , onDeAct:

```
[MyVariable] call MyFunction;
```

or

```
MyFunctionRetVal = [MyVariable] call MyFunction;
```

Functions may or may not return a value.

```
MyFunctionRetVal = [MyVariable] call MyFunction;
```

Returns a value

```
[MyVariable] call MyFunction;
```

Does not.

All code lines needs to be closed with “;”

Except the last line which returns the value from the function, which mustn't have the “;”

e.g.

```
_myVariable = "Returns Hello";  
//Return value  
_myVariable
```

Functions cannot be paused with sleep as procedures can.

Functions will halt the script it was started from until it is complete.

For example:

```
while{alive _myGuy} do {  
    _myValue = [_myGuy] call MyFunction;  
    /*more code here  
    .....  
    */  
    hintSilent format ["Value:%1",_myValue];  
    sleep 1;  
};
```

hintSilent will not be reached until MyFunction has completed its execution.

Variables

Variables are defined and initialized by setting them with a default value.

```
_myString = "";  
_myArray = [];  
_myNumber = 0;  
_myBoolean = false;  
_myObject = objNull;
```

Or directly initialized by the value of something else, for example:

```
_myString = format ["%1",name player];  
_myArray = _someOtherArray;  
_myNumber = (1 + 1);  
_myBoolean = ((side player) == west);  
_myObject = player;
```

In a script these variables live in something called a “scope”, the block or control structure. Blocks are the code within {}, or the statement followed by “;”

e.g.

```
hint "Hello";
```

Variables defined with underscore “_” are local in the script, variables without are global and can be read from, and or set by other scripts.

Example:

MyProcedure.sqf

```
MyVariable = "Will be overwritten"; //Can be read or set from other scripts
_myFunction = compile preprocessfile "MyFunction.sqf";
_fromOther = [] call _myFunction;
//Shows "Returns Hello, Global Hello" on the screen
hint format ["%1, %2", _fromOther, MyVariable];
```

MyFunction.sqf

```
_myVariable = "Returns Hello"; //Local to this script only
MyVariable = "Global Hello"; //Can be read or set from other scripts
//Return value
_myVariable
```

There are also public variables, which are your global variables, set by:

```
publicVariable "MyVariable";
```

This will broadcast the value of “MyVariable” to the other computers on the network in a MP-environment.

Locality and MP environments will not be covered in this guide as of yet.

Do not use variable names which are protected or restricted to BIS commands, for example.

```
magazines = (magazines player);
```

More on variables

The different types of variables available are called “data types”

There are different types of them as you have seen, and below is a brief explanation of them and examples of how to use them.

Strings

As you’ve seen in this document, there are different ways of displaying these strings, and formatting them.

The basic way of formatting a string is to use the command `format`

```
_myString = format ["Name: %1\nWeapon: %2", (name player), currentWeapon player];
```

`_myString` now holds the player name and the current weapon used in two rows.

e.g.

Name: Taurus

Weapon: M16

The `%[n]` is used as a reference to the values after the first “,”.

New lines are created with “\n”.

You can concatenate or “add” one string to another like this.

```
_myString = "Man " + " far" + " away" + "from us!";  
_myString now holds “Man far away from us!”
```

And to display it on the screen, the command *hintSilent* has been used.

```
hintSilent = format ["%1", _myString];
```

There are three types of “hints”:

- hint – displays text with a “ping”-sound
- hintSilent – displays the text without the “ping”-sound
- hintC – displays a confirm dialog, then shows the text in the upper right corner.

Also the command *sideChat* has been used.

Do mind that the new line “\n” does not work on any type of chat-command.

Both “hint” and “Chat” are powerful tools when debugging your script to see different values of variables.

Arrays

Some of the BIS commands return arrays, for example (*crew _myVehicle*)

Which will return the current crew in the vehicle, both cargo and for example the driver and gunner.

The BIS: Wiki for the specific command will tell which data type the command returns, if anything is returned. Take a look at [crew](#) now.

The values or content of arrays are called “elements”.

Elements in the array are accessed by indexes where the first element is at index zero (0).

As you’ve seen in the examples to retrieve the inputs to the scripts:

```
_myGuy = _this select 0;
```

Selects the first element at index zero, if more arguments are sent into the script:

```
_myGuy = _this select 0;  
_myStr = _this select 1;  
_myNum = _this select 2;
```

The argument array could have contained: [MyGuy1, "Hello", 63]

You may also of course define your own arrays containing variables.

```
_myArray = [MyGuy1, MyGuy2, MyTank1];
```

You can add one array to another, in the same way as you did with the string.

```
_myArray = [MyGuy1, MyGuy2, MyTank1];  
_myBigArray = _myArray + [MyGuy3, MyGuy4, MyTank2];  
_myBigArray now holds “[MyGuy1, MyGuy2, MyTank1, MyGuy3, MyGuy4, MyTank2]”
```

As you see the array retains order of insertion.

You can subtract elements from arrays too.

```
_myArray = [MyGuy1, MyGuy2, MyTank1] - [MyGuy2];  
_myArray now holds “[MyGuy1, MyTank1]”
```

Arrays in this script engine can hold different data types, for example.

```
_myArray = [MyGuy1, "Meaning of life:", 42];
```

Use this kind of “multi data type”-array with care though, as comparing the elements will be like comparing pears with apples.

If you want to change an element in your array you use the command *set*

```
_myArray = [MyGuy1, MyGuy2, MyTank1];  
_myArray set [2, MyTank2];
```

This will change the second index of the array to “MyTank2”

_myArray now holds “[MyGuy1, MyGuy2, MyTank2]”

Arrays are useful because you can iterate, or loop over each element in the array.

Loops are covered in the section “Control structures”.

The current size of arrays can be returned by using the command *count*

```
_myArray = [MyGuy1, MyGuy2, MyTank1];  
_arraySize = count(_myArray);  
_arraySize is now 3.
```

You can also find the specific index of an element in the array, using the command *find*

```
_myArray = [MyGuy1, MyGuy2, MyTank1];  
_myIndex = _myArray find MyGuy2;  
_myIndex is 1, i.e. the second element at index 1.
```

When the element is not found -1 will be returned.

```
_myArray = [MyGuy1, MyGuy2, MyTank1];  
_myIndex = _myArray find MyGuy3;  
_arraySize will be -1.
```

There is also a possibility to check if an element exists in the array, using the command *in*

```
_myArray = [MyGuy1, MyGuy2, MyTank1];  
_myBoolean = MyGuy2 in _myArray;
```

Returns *true* if it is in the array or *false* if not.

Numbers

Numbers can be any numeric value, for example.

```
_myNumber1 = 242;  
_myNumber2 = 0.42;  
_myNumber3 = -22;  
_myNumber4 = -2.30;
```

Numbers can naturally be subtracted, multiplied, divided and added to or with any other numeric value.

Watch out for zero divisors though!

Objects

I’ve decided to group this data type to one, this hold the rest of the data types, for example.

```
_myGroup = (group player);  
_myStreetLamp = nearestObject [(getPos player), "StreetLamp"];  
_myHandler = [player] spawn MyProcedure;
```


Magic variables

There are some more data types which aren't covered by any of the other data types.

The "magic" ones, these are.

- `_this` – which return arguments to all kinds of scripts, you have seen "`_this`" used many a times in this guide.
- `This` – which holds the current object, for example, unit init lines where "`this`" will return the unit (object) itself.
- `ThisList` – which return an array of objects that are currently fulfilling the trigger condition. For example, "`BLUFOR present`", will return all BLUFOR units in the trigger.
- `_x` – which return the current element in a "`forEach`"-loop.

Advanced usage of variables

There is one interesting aspect of how to set variables.

Objects can have their own variable space, and these are set by `setVariable`

```
_myUnit setVariable ["UnitMags", (magazines _myUnit), false];
```

This will create a variable called "UnitMags" holding the current magazines equipped by the unit and assign it to the unit variable space.

The last element is a Boolean (true or false) if the variable should be broadcasted over the network.

To retrieve the value of this variable you use the command `getVariable`

```
_unitMagazines = _myUnit getVariable "UnitMags";
```

Even more information about variables

If you still aren't satisfied you could find more about the different data types in the Appendix: Data types.

How to compare these data types with each other please see the section Operators in this guide.

Control structures

You've been using most of these already, below is a brief explanation of the most commonly used control structures.

if-statement

To be used when some value needs to be checked in order for a code block to run.

```
if(condition) then{
    //Do stuff.
};
```

The condition must return a Boolean value (true or false)

For example:

```
if(_myNumber == 10) then{
    //Yep, its ten!
};

if(alive _myGuy) then{
    //MyGuy is still kicking!
};
```

These conditions use operators, which will be explained later.

if-else statement

To be used when the condition in the if-statement is not met, for example:

```
if(_myNumber == 10) then{  
  
    //Yep, its ten!  
}else{  
    //Number is not ten!  
};  
  
if(alive _myGuy) then{  
    //MyGuy is still kicking!  
}else{  
    //MyGuy is dead :(  
};
```

else-if statement

Use this if you want to check more than one value depending on the original condition.

For example if one guy is dead, check if the other one is still alive.

```
if(alive _myGuy) then{  
  
    //MyGuy is still kicking!  
}else{  
    //MyGuy is dead :(  
    if(alive _myGuy2) then{  
        //Well one is still alive  
    }else{  
        //Both is dead!! :'(  
    };  
};
```

switch-statement

if-statements can be a mess at times, as you can see from the “else-if statement” above, especially if you are not checking Boolean(true or false) values.

Then you can use the switch-statement instead.

```
switch (variable) do {  
    case value1: {  
        //Do stuff  
    };  
    case value2: {  
        //Do stuff  
    };  
    //As many cases as you want, case values should be unique though.  
    default {  
        //Do stuff  
    };  
};
```

For example if you want to check the side of a current unit, instead of having several if-else statements, for example:

```
switch (side _myGuy) do {
    case west: {
        hint "West!";
    };
    case east: {
        hint "East!";
    };
    case resistance: {
        hint "Guerilla!";
    };
    default {
        hint "Civilian or not defined!";
    };
};
```

As you see the “default” block is the code to run if neither of the above cases is fulfilled.

for-loop

This is a loop which iterates for a specific number of times.

```
for [{variable}, {condition}, {increment}] do {
    //Do stuff until condition is met.
};
```

For example, this can be used as a counter

```
for [{_iX = 1}, {_iX <= 10}, {_iX = _iX + 1}] do {
    //Displays 1 to 10 on screen.
    player sideChat format ["%1",_iX];
    sleep 1;
};
```

This loop can also be used to iterate over arrays, as such.

```
for [{_iX = 0}, {_iX < (count _myArray)}, {_iX = _iX + 1}] do {
    player sideChat format ["Element at index: %1=%2",_iX,_myArray
select _iX];
};
```

forEach-loop

The forEach-loop is the most common way for iterating arrays.

```
{
    //Do stuff
}forEach _myArray;
```

Example of how to iterate all units in your group:

```
{
    player sideChat format ["%1",name _x];
}forEach (units (group player));
```

Note the use of the magic variable “_x” here.

Even more on control structures

As I intentionally left out some of the control structures, Please find more information in the related article under the Appendix regarding “Control structures”.

Operators

There are several types of operators in this scripting language.

I will only be mentioning the ones used in this guide.

Logical Operators

These are used to compare Boolean (true or false) values.

Terms:

"!" means "not"

"&&" means "and"

"||" means "or"

These logical operators return Boolean values.

Examples:

```
_isDead = !(alive _myGuy); //True if unit is dead, false if alive.  
_bothAlive = ((alive _myGuy) && (alive _myGuy2)); //True if both is alive,  
false if one of them dies.  
_anyAlive = ((alive _myGuy) || (alive _myGuy2)); //True if any of them is  
alive
```

Comparison Operators

These are to compare the other variables, i.e. non Boolean variables.

Terms:

"==" means "equals"

"!=" means "not equals"

"<" means "less than"

">" means "greater than"

"<=" means "less or equals"

">=" means "greater or equals"

These comparison operators return Boolean values.

Examples for numbers:

```
_myNumber = 42;  
_isFourtyTwo = (_myNumber == 42); //True  
_isNotFourtyTwo = (_myNumber != 42); //False  
_isGreaterThanFourtyTwo = (_myNumber > 42); //False  
_isLessThanFourtyTwo = (_myNumber < 42); //False  
_isGreaterThanOrEquals = (_myNumber >= 42) //True  
_isLessThanOrEquals = (_myNumber <= 42) //True
```

Even more on Operators

There are more to them than I wrote here, so please read more in the Appendix about Operators.

When Do I Need Scripting?

Be careful: Scripting isn't a solution to everything.

The first thing to ask yourself is “Am I absolutely, positively sure this cannot be done using just the editor?” The goal with scripting is to create processes that can't be done otherwise. Scripting does use system resources, poorly written scripts can affect game play and/or performance. So, it pays to be sure you have learned as much as you can about how the editor works and you understand its capabilities and limitations.

The second thing to ask before you start scripting away is “Will players even notice and/or use the action or feature I would like to implement?” It may seem silly, but just because it can be done does not always mean it should be.

The third step (and after the first two are out of the way just may be the hardest step, especially for people new to scripting) is to try and determine if what you want to do can be implemented with the scripting language. ([From the wiki](#))

Scripts help you doing things which aren't possible to do with the editor.

For instance, spawning or moving a unit to a dynamic location at any given time in your mission. Making distance calculations, determine the current altitude of an aircraft, to name a few.

Or lines of code which simply are too long to put in the unit init lines or trigger condition and or onAct, onDeAct.

If you write your script code to be generic you should only need one script per task.

With scripting, missions can be taken to the next level of re-playability without preset locations of waypoints, and you as the mission maker doesn't need to re-make the whole mission in the editor each time.

Best practices

Variables should be set by using the “private” command, this to prevent variables to be accidentally overwritten by another script executed from within the starting script.

The use of the private command is shown in the below example which also describes the block and scope.

```
//Scope 1
private ["_myVariable", "_myString"];
/*_myHandler = [<string>] spawn MyProcedure*/
_myVariable = _this select 0;
/*defined here, and set by either result from the if-else statement.
If not defined here, the last line would failed with "variable not
defined"*/
_myString = "";
if(_myVariable == "Hello") then{
    //Scope 2
    //Code block 1
    _myString = "He said hello!";
    _myBlockNumber = 0; //this variable cannot be read from code
    outside this block or scope...
} else{
    //Scope 3
    //Code block 2
    /*i.e. this wouldn't work
    hint format ["%1", _myBlockNumber];*/
    _myString = "He didn't say hello, what manners!";
};
//Will show "He said hello" or "He didn't say... "
hint format ["%1", _myString];
```

The private command has been intentionally left out from the “My first” examples.

Global variables, procedures and functions should be defined in the “init.sqf” file as you did in the “My first function” example.

Unless you decide the procedure or function is only to be used ones.

In procedures when you use loops, add a `sleep X;` in the loop block, this to prevent the game to grind to a halt, and also let the CPU do other things than processing your loop. In cases where you by accident made a scripting error, and forgot this sleep, the only way is to kill the Arma 2 process, this because there will be no time to fetch the “Esc”-click, and you cannot exit the game by normal means.

e.g.

```
while{alive _myGuy} do {
    //My code here and here and etc
    sleep 0.125;
};
```

If you want to sleep a `waitUntil`-command as described on the “wait until script completes” in this guide.

```
waitUntil{
    sleep 0.125;
    (myCondition)
};
```

The last line is the return value, and as for function return values, it should not have “;”

Try to make your code easy to read, if you put parenthesis around the code it will be more readable and easier to understand, and even let the script engine make an easier task of understanding what you really want.

For example, selecting elements in an array:

```
_myXPos = getPos _myGuy select 0;  
_myYPos = getPos _myGuy select 1;  
_myZPos = getPos _myGuy select 2;
```

getPos returns an array, so putting parenthesis around the complete command.

```
_myXPos = (getPos _myGuy) select 0;  
_myYPos = (getPos _myGuy) select 1;  
_myZPos = (getPos _myGuy) select 2;
```

Complex calculations, (just making something up here):

```
_myNumber = _myNumber1 * 1.34 + 36/4;
```

The script engine is smart to parse out what to add and multiply, but it may not do exactly what you want, to enforce this, again use parenthesis.

```
_myNumber = (_myNumber1 * 1.34) + (36/4);
```

Complex conditions:

```
_myBoolean && _myOtherBoolean && _myNumber == 42 || !alive _myGuy
```

What would be compared in the above condition?

Use parenthesis to group the conditions.

```
((_myBoolean && _myOtherBoolean) && (_myNumber == 42)) || (!alive _myGuy)
```

Resulting in: (all these three are true) or (unit is dead)

Use procedures for heavy scripts that will run alongside the game, and or include loops of any kind. This might be track unit movements, check if values are true and so on.

Use functions to handle return values or where you want to halt the script until a condition is met or to make quick calculations.

Have your mind set up on the script to be generic, you might want to use the script in another mission later or when you have made a very good script and want to share your script with the community.

Using argument variables rather than hard coded values means that you don't have to rewrite the script for each mission.

Having the "My first function" example in mind, where you sent in the unit (you the player) as an argument to the script.

What if you wanted to check if an AI controlled unit was in the vehicle instead?

The AI unit is called MyGI_1, in the "My first function" example you'd simply change the player argument.

```
player sideChat format ["%1", [MyGI_1] call MyFunction]
```

And if you want to check both you'd add another script line to the triggers onAct.

```
player sideChat format ["%1",[player] call MyFunction]; player sideChat  
format ["%1",[MyGI_1] call MyFunction]
```

Do mind the “;” between the two, as “;” is used to separate them as you'd do with new lines in scripts.

You should also have in mind the size of your script, each script running will use CPU and RAM, so narrow it down where you can.

Also keep in mind the possible memory usage of your variables.

For example, in a script:

```
_myLongString = "The ships hung in the sky in much the same way that bricks  
don't."; //65 byte  
hint format ["Quote:\n%1",_myLongString];
```

Having ten scripts running with this script uses 650 byte of memory.

Instead if you want to use this string in a static way, consider having it as a global variable in “init.sqf”

```
MyLongString = "The ships hung in the sky in much the same way that bricks  
don't."; //65 byte
```

And then in the script:

```
hint format ["Quote:\n%1",MyLongString];
```

Document your code.

You will thank yourself by doing so when you come back to your old scripts some weeks or even months later.

Naming conventions

This is up to you, but it is good to in some way state in the script file name if it is a function or procedure, for example as used in this guide “MyFunction.sqf”, “MyProcedure.sqf” or “FNC_CheckInVehicle.sqf”, “ShowCurrentSpeed.sqf” with abbreviations and shortened description of what the script does.

Or place the scripts in different folders.

e.g.

Functions

Procedures

Appendix

Related wiki articles used in this document

[Control structures](#)

[Data types](#)

[SQF Syntax](#)

[Operators](#)

Example of Crew command wiki article

Bohemia Interactive

navigation

- Main Page
- Community portal
- Current events
- Recent changes
- Random page
- The Village Pump
- Help

search

Go Search

toolbox

- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link

crew

Editors, please check *Policy: Scripting Command Page Syntax*.

1.00 **AG** **Global**

Click on the images for descriptions

Introduced in

Game: Operation Flashpoint

Version: 1.00

Description

Description: Returns the crew of the given vehicle.

Syntax

Syntax: **Array** = crew vehicleName

Parameters: vehicleName: **Object**

Return Value: **Array** - An array with all units in the vehicle is returned.

Examples

Example 1:

```
player in (crew _tank)
```

Additional Information

Multiplayer: -

See also: **commander, driver, gunner**

Annotations:

- Command locality restrictions, if any.
- Click links for explanation of data types.
- Click links to see related commands

About the author

I'm known as "Taurus" on the BIS official forums, and I've been scripting this game-engine since OFP, mostly for ArmA.

I did not find any SQF guide and the existing guides concentrates on SQS and, or the editor itself.

As it has come to my attention that this kind of guide was needed I decided to write one.

I do hope this guide come of great use.

Feedback, requests for new versions of this guide, and or if something is unclear, please send them to:

plastrader_242@hotmail.com

Put "ArmA 2 scripting" in the subject.

Thanks for reading.

Happy scripting and "Don't panic!"

/ Taurus

Versioning

Version 1: First version of this document.

Version 2: Updated sections, formatted the code, added the “When Do I Need Scripting?” section, added wiki article about operators, added sub section “More on variables”.

Version 3: Renamed document to “Guide”. Added section “Examples”, updated “Best practices”, pretty much rewrote the whole thing.

Version 3.1: Updated examples, with (alt+tab), made some changes in the text.

Version 3.2: Updated usage of arrays, best practices “memory usage in scripts”, updated “crew article”-picture, with explanation boxes.

To-do:

Add example using several instances of procedures.

Add section about debugging.

Add section on how to prematurely exit scripts.

Add section about “inline functions” and “inline procedures”