Basic Scripting Tutorial

Operation Flashpoint Resistance's Scripting Language explained

Version 0.90

Written by Daddldiddl
(daddldiddl@gogodot.net)

Introduction

Many aspiring mission makers use scripting, but lack any basic knowledge of programming – necessary to understand the restrictions and possibilities the scripting language for Operation Flashpoint poses - thus making it hard for them to learn from other scripts and to script new stuff instead of just copying things they have seen somewhere else. I will not go into explaining all the commands and offering solutions for all problems, so if you see any commands in my examples that you do not understand, just consult one of the references recommended on page 18.

If you think something's missing, or an explanation or example is wrong or hard to understand, just mail me and I'll try to improve it. Questions for help with your own scripts will **not** be answered – please understand that I got a life besides OFP. Asking your questions in one of the many scripting forums (for example at www.flashpoint1985.com or www.ofpec.com) will bring you in contact with more proficient mission makers than myself and get you an answer more quickly and more thoroughly anyways. When visiting these sites please use the search function – most questions have already been asked and answered more than once.

The most recent version of this document will always be available at my OFP website at http://daddldiddl.de (download section, "Tutorials"). Any commercial distribution of this document is prohibited by the author.

Thanks to all the people participating in the official forum, to Lester for his great command reference, to Mr. Murray for his Editing Anleitung (you showed me the way, man!), to Bohemia Interactive Studio for providing us with this great game, and to all members and friends of my squad SES (http://www.suicidesquad.co.uk) for being such a great bunch of people to play with.

Daddldiddl (aka Joltan),

August 20, 2003

From Problems to Scripts

So you have no clue about scripting, but you want to learn it? A script is a program that is interpreted upon execution by another program (the so called ‘parser’ – in our case OFP). For the parser to understand what we want him to do, we have to write our instructions using the commands and program structures it knows.

The first thing to do is always to analyze your problem and then divide it in logical steps, define the decisions to be made and the input/output needed. The resulting solution or ‘algorithm’ is what you then formulate in the scripting language.

A simple (non-OFP) example:

Even just cooking an egg is quite a complicated process considering the many steps needed to get it right:

First we have to get a pot and fill it with water, then the heater must be lit. Then we put the pot on it and wait until the water is boiling.

Once it is hot we put the egg in and then wait for the time needed to cook the egg just as we want it. Then we take the eggs out of the water and turn off the heater.

The above is the algorithm or ‘receipt’ telling our parser (in this case the cook who has to prepare the egg) what we want it to do.

Whenever we want to solve something in OFP with a script, we have to come up with such a ‘receipt’ and formulate it in the the game’s scripting language. For Operation Flashpoint scripts can be written as simple text files with Notepad or any other text editor.

An every-day algorithm (©1987 SYBEX-Verlag GmbH, Düsseldorf)

For Operation Flashpoint the command reference that is available at the Bohemia Interactive Studio (BIS) website at www.flashpoint1985.com (look in the editing section) is what you need to get a list of all available commands and the supported data structures. It is a bit short in its explanations, so in addition check the available tutorials and unofficial (and unfortunately mostly incomplete) references at the Operation Flashpoint Editing Center (www.ofpec.com) and other editing websites (see *References* section on page 18).

This scripting tutorial will mainly focus on describing the structures you then can fill with the necessary commands to solve your specific problem.

Basic Structures

Scripts

Basic Script Structure

A script is a simple text file. You can edit it with Notepad or any other text editor. If you want you can also use one the many script editors available for OFP, but they are not really needed (they offer additional stuff like included command references or automatic color coding of the source – don't expect the latter to work properly for anything complicated).

- The script usually contains one instruction per line.
- Some expressions (like command strings can spread over several lines – see chapter *Functions*, page 12)
- It terminates on encountering 'exit' instruction. This should always be placed at the end of a script, but can also be used in conditions to exit earlier.

Example for a simple script:

```
_distance = unit1 distance player
_name = name unit1
_output = Format[“%1 is %2 meters away”,_name,_distance]
hint _output
exit
```

In the above example the distance from another unit to the player is measured (1st line), then the units name is derived (2nd line), the two variables *(_distance* & *_name*) are included into a text (3rd line) and then this text is displayed on the screen (4th line). Finally the script exits.

Executing a Script

You can start a script from a trigger, a unit's init line or from another script. You can start a script with parameters, but no value will be returned – for that you need to define a function (later more on that) or use global variables.

Basic syntax: `<parameter> exec “scriptname.sqs”`

The string at the end is the filename of the script. The game assumes the script to be found in your mission's root directory (where the mission.sqm is saved, too). You can also save it in a subdirectory of this folder, but you would have to add the path to the string (see examples below). Note that the file extension is '`.sqs`'. While you could use virtually any extension you like, this is the standard that everybody understands (better if you may send your mission to someone else asking for help), and it will also help you to keep your mission's folder from sinking into chaos!

The parameter may be a single value, variable or object name. Only one parameter can be used. If you need to call the script with more than one value you need to use an array as parameter. If you do not need to call the script with any parameters use an empty array.

Examples:

```
[] exec “myscript.sqs”
player exec “myscript.sqs”
_targetpos exec “myscript.sqs”
_targetpos exec “scripts\myscript.sqs” (in subdirectory scripts!)
[pilot1,“LaserGuidedBomb”,_targetpos] exec “bombrun.sqs”
```

To access the parameter from within the script we can use the reserved variable `_this`.

Example: `player exec “myscript.sqs”` *(script call)*
⇒ Script: `RemoveAllWeapons _this` *(direct use as local variable in the script)*

If an array was used as parameter, we can read its values using the select instruction:

Example: [player,[x,y,z]] exec “myscript.sqs” *(script call)*
⇒ Script: `_unit = _this select 0` *(inside the script the array gets read and*
`_position = _this select 1` *its values assigned to local variables, so*
`_unit domove _position` *they can be used in instructions)*

Variables

Variables are just names for objects and values. They are always references for something. The values or the objects they represent may be changed but the name stays the same.

For example you need the number of units in the player’s team, but you don’t know that number in advance (either because units leave/join the group during the game or because it’s a multiplayer mission where not all slots might be occupied). Of course you could query the number of units each time you need it, but that would be a lot of typing. By querying once and then using a variable to reference to the result of the first query, you don’t need to type as much and you also save processing time as the query requires the game to do much more work to come up with the result than just to check its memory for the value.

Variable Types

There are numerical, string, boolean, array and object variables:

- **Numerical variables** are always real numbers (i.e. they can have decimals).
- **String variables** are basically text. They may contain any character, and when values are assigned to them they have to be in parenthesis.
- **Boolean variables** have only two possible values: they can be *`true`* or *`false`*.
- **Array variables** are a list of variables within a variable, see *page 5* for more details
- **Object variable** these reference to an object (like a specific unit, a certain vehicle, etc.). The names you can give units in the mission editor are an example of such a variable.

It is not possible to calculate with string variables, even if they contain numerical characters only. You would have to convert them to numerical variables first (but that goes far beyond the aim of this tutorial). If, for example, we have 3 variables `A`, `B` and `C` with `A` representing the value of 10, `B` representing the value of 30 and `C` representing the string “15”, then we could multiply `A` with `B`, but we wouldn’t be able to divide `B` by `C`.

You can name your variables with names as long as you like (there is some limit, but it’s way more than 8 letters), and you can use letters (a to z) and numbers in the name (note the name has to begin with a letter). Underscores can be used to make long variable names more legible (for example `number_of_players_in_team1` would be a valid variable name). The name may not contain spaces or other special characters apart from the underscore. The variable names are not case sensitive – so `NumberOfPlayers` is the same as `numberofpLayers.`

Note that starting a variable name with an underscore has a special meaning – it makes the variable local. See the chapter *Global vs. Local Variables* on page 6 for more information. Also there are some reserved variable names (like `player, _x, _this` and some others).

Assigning/Computing a Variable

Just place the name of the variable first, followed by a single equal sign and then the value/object it should reference to.

Examples:

Numerical:	`A = 10 + 12`
	`B = 10 + A` *(with `A` being a numerical variable)*
	`N = count (units group player)`
String:	`T = “this is a string”`
Boolean:	`X = true`
Object:	`O = vehicle player`
Array:	`A = [14,23,“some string”]`

Some Simple Math in OFP

Addition & substraction:	`a + b - c`
Multiplication & division:	`(a * b) / c`
Taking the square sum:	`x^2`
Taking the square root[1]:	`x^(1/2)`
Random number:	`random (x)`

Arrays

An array is a list of values. They can be assigned to a variable name as usual, but have to be put in “[…]” with the values separated by commas. To read a value from the array you need to `select` it by it’s position within the array, starting with 0 for the first entry and then counting up. Different kinds of values (numerical, boolean, strings, arrays) can be used side by side within an array. In the example b would result to be numerical with a value of 72, as it’s the third entry (= position 2).

Example: `b = [23,true,72,”bla”] select 2`

It is possible to have cascading arrays (one array within another, in which case you have to use several select commands to read the values:

Example: `b = ([23,97,”bla,[“blabla”,15],28] select 3) select 1`

⇒ `b` *= 15*

You can count the number of entries within an array with the `count` command, you can check if a value/object is present in an array with the `in` query, you can remove or add entries to an existing array, and you can change an entry by overwriting it with the `set` command.

---

[1] as $\sqrt[n]{x} = x^{1/n}$ you can use the same ‘trick’ to take the root of the 3rd, 4th or any other grade

Examples:

`b = count [23,true,72,″bla″]` ⇒ `b` = *4*
`b = count [23,97,″bla,[‶blabla″,15],28]` ⇒ `b` = *5*
`b = count ([23,97,″bla,[‶blabla″,15],28] select 3)` ⇒ `b` = *2*
`b = ‶_x > 10″ count [20,12,4,18,7]` ⇒ *b = 3, only entries > 10 are counted*
`b = [20,12,4,7,18,7] - [12,7]` ⇒ `b` = *[20,4,18]*
`b = [20,4,18] + [29,15]` ⇒ `b` = *[20,4,18,29,15]*
`b = [20,4,18] set [1,″bla″]` ⇒ `b` = *[20,”blah”,18]*

Formatting Strings

Sometimes it is not possible to know exactly what you want to put in a string in advance. For example you'd like to output a string telling the player how far he is from his team leader. You can't know the distance in advance, but you could calculate it. Unfortunately the result will be numerical and just telling him the plain number would look a bit dull. So we use the `format` command to combine different values to one string.

Example: `b = format [‶%1, you are %2m away″,(name player),(player distance (leader group player))];`

If player Joe were 250 meters from his leader, the resulting string would be *"Joe, you are 250m away!".* The array expected by the format command always contains first the output string with place holders for the values to be inserted upon execution (starting with %1 and then counting up). After this the values are listed, separated by commas. You may use variables or even compute the values on the spot (as in the example above).

Note: String variables may contain parenthesis, too – but to distinguish the parethesis that belong to the string from those that mark the string's begin and end, all internal parenthesis are doubled.

Wrong Example:	`a = ‶and he said: ″damn!‶, and went home″`	
`a` would be interpreted as:	`'and he said: '`	⇒ *String No. 1*
	`damn!`	⇒ *unknown instruction* ⇒ *ERROR!*
	'and went home'	⇒ *String No. 2*

Correct Example:	`a = ‶and he said: ‶‶damn!″″, and went home″`
`a` would be interpreted as:	`'and he said: ‶damn!″, and went home'`

Global vs. Local Variables

Now that we know how variables are assigned in scripts the question arises if we can use the variables outside the script (for example calculate a value within a script and then have a trigger reacting if the variable's value exceeds a certain amount)? And could this cause conflicts when running several scripts that use the same variable names at the same time (for example running two or more instances of the same script)? What if we use the same variable name in one script for a string variable and in another for a boolean variable?

The answers depend on whether you use global variables or local variables in your scripts:

- **Global variables** can be queried from outside the script. If one script assigns a value to a variable and another script changes this value, then the first script will use that changed value.
  This may be useful in some cases - for example setting a boolean variable to false,

then waiting for another script to set it to `true` again before continuing, or using a global variable to make the computed values intentionally available to other scripts. But in many cases this leads to conflicts – and if your mission uses many scripts, then you may easily loose track of which variable name you used in which script.

- **Local variables** can not be queried from outside the script - they are 'local' to the script. Therefore there can be no conflicts with another script using the same variables or variable names.
  Local variables have their names starting with an underscore. For example `_n` is a local variable while `n` would be a global variable.

As a basic rule you should only use global variables in scripts when you need to have the variable interacting with the rest of the game (to toggle triggers, or to have it changed or queried by other scripts). For everything else use local variables.

Note that global/local is in relation to the script that uses the variables, and not in relation to whether a variable is available only on the local computer or on all computers in a multiplayer game. **Variables – whether they are 'local' or 'global' – always only exist on the machine where the script is running**. They are not automatically transferred over the net. To distribute/synchronize a value over the net you must use the `PublicVariable` command[1].

Conditions

Syntax for Conditions

Conditions are important for triggers and scripts. With triggers you just write the condition in the condition field, while you have two possibilities with scripts:

- *`?(<condition>):<single command>`*

  This only allows for a single command to be executed if the condition is `true`. This can be circumvented by using the `goto` command to jump to another location in the script. So this syntax is great for the simple, conditional execution of single commands or for branching within scripts.

  Example: `?(a > b):unit1 sidechat "a is bigger then b!"`

- *`if (<condition>) then {<command string>;};`*
  or
  ```
  if (<condition>) then {<command string>}
                        else {<command string>};
  ```

  This allows for multiple commands to be executed in case the condition is `true`, and allows for the execution of an alternative set of commands in case the condition is false (optional). Note the different syntax: within the so called command strings (the "{…}") each command must be terminated with a semicolon, and another one must be placed at the end of the if-expression. The commands may be all fit in one single line or be spaced over several lines for easier reading.

---

[1] The only exception is the *init.sqs* script if the variables are defined before the mission starts (i.e. before any command is executed that causes the script to wait for the mission to start. That makes it possible to use random values that are the same on all clients in multiplayer games.

Example:

```
if (a == b)
  then {
    unit1 sidechat “a equals b!”;
    unit2 sidechat “Thats nice!”;
    }
  else {
    unit1 sidechat “a does not equal b!”;
    unit2 sidechat “That’s unfair!”;
    }
  ;
```

`If-then(-else)` conditions can be used in conjunction with the `foreach` command to apply instructions only to those members of an array that meet a certain condition.

Example:

`“if (not alive _x) then {deletevehicle _x}” foreach units group1`

⇒ *this will delete all dead members of group1 from the map*

Note: the `if-then`(`-else`) syntax was introduced with version 1.85, and will therefore not work with older versions. Also the command string poses some restrictions: within a command string it is not possible to use jumps and loops or to halt the script temporarily (see the chapters *Loops and Jumps* on page 9 and *Taking a Break* on page 11).

Comparing Values

So what possibilities do we have to compare values and setup our conditions? For a list of operands see the next page. Conditions can be formulated to form complex expressions using brackets and the ‘`and`’/‘`or`’ operands. Note that the ‘`or`’ is not exclusive, meaning that it is not only `true` if either condition is `true`, but also if both conditions are `true`!

Besides using conditions like ‘bigger than’ or ‘equal to’ (see syntax below) any boolean variable can be used instead of a condition, as it also returns either `true` or `false`. So `?(enemydead):player sidechat “I win!”` would execute the command if the variable `enemydead` was set `true`.

`a == b`	Is `true` when `a` is equal to `b` Note: do not use a simple ‘=’, as this is only used to assign a value to a variable!!!

`not <condition>`
or
`!<condition>`
	Inverts a condition, so it is `true` when the condition is not met.
	Example: `(not alive player)` would be `true` when the player is dead. This could also be expressed as `!(alive player)`

`a < b`	`a` smaller than `b`
`a > b`	`a` bigger than `b`
`a <= b`	`a` smaller or equal to `b`
`a >= b`	`a` bigger or equal to `b`
`a != b`	`a` not equal to `b`
`<condition> AND <condition>`	Is only `true` when the first and the second condition are both `true`
`<condition> OR <condition>`	Is `true` when either the first or the second or both conditions are `true`
`a in <array>`	Is `true` if `a` exists at least once in the array. *Example:* *`“bla” in [“blubb”,15,”bla”,”bleh”]`*

Loops and Jumps

Sometimes it is not enough to have the parser working through the script from start to end. You might have repeated tasks that would force you to either restart the script over and over or to type the same things again and again – especially problematic if you don't know in advance how often you need to repeat the task. Or you might have parts of your script that should only be executed under certain circumstances – and then maybe should be executed from different parts of the script. For this you need loops or jump. Functions are another possibility, but these will be discussed in a later chapter.

Jumps

Jumps are very simple – you set a marker at a certain position in a script, and when you need to go there you just use the `goto` command to jump to this position. This way you can jump over parts of the script that you don't need at the moment, or create loops.

The position must have a unique name (within the script). If the same marker exists twice then the script will always jump to the first marker from the top. A marker is placed by writing a '#' followed directly (no space in between) by the name. As with variables these names are not case sensitive and the same naming guidelines apply. To jump to the position you use `goto` command followed by the name in parenthesis (for example `goto "marker1"`)

Example:

```
?(a > b):goto "marker1"
<…some lines of code…>
#marker1
<…script continues…>
```

If `a` is bigger then `b`, then the lines between the condition and the marker will not be executed, but the script will continue at the position of `marker1`.

Sometimes you might want to set the target of the jump more variable – when you look at the `goto` command, you see that it expects a string as input. Therefore you can substitute the fixed target information with a string variable that you defined before. That way you can jump to another position in your script and then return to where you started the jump, or you can randomly jump to certain parts of your script!

Example:

```
_jump=["jump1","jump2","jump3"] call RandomSelect
goto _jump

#jump1
  <…some lines of code…>
  goto "end"
#jump2
  <…some lines of code…>
  goto "end"
#jump3
  <…some lines of code…>
  goto "end"
#end
```

This example uses a function (a preloaded subroutine, see page 12) called *RandomSelect* to randomly choose one of the three strings from the array and then go to the respective section of the script (using the selected string as the target location).

Loops

A loop is created by simply putting the jump marker before the `goto` command in a script:

Example:

```
#marker1
<…some lines of code…>
goto "marker1"
```

This will repeat the code between the marker and the `goto` command until the mission ends. Of course you usually want to have an abort condition. You can either make the `goto` conditional or put another marker behind the `goto` so you can jump there from any line within the loop.

Examples:

```
#loop
  _height=getpos player
?(_height < 10):goto "loop"
player sidechat "I'm flying!!!"
```

or

```
#loop
  _height=(getpos player) select 2
  ?(_height > 10):goto "endloop"
  player sidechat "I wanna fly… (whine!)"
goto "loop"
#endloop
player sidechat "I'm flying!!!"
```

Both loops will repeat until the player is higher than 10 meters [1].

Sometimes you know exactly how often you want to repeat a loop. To set the number of repetitions for a loop we need one local variable and an abort condition based on this variable.

Example:

```
_n = 5
#loop
  _n = _n - 1
  player sidechat (format ["%1 loops to go!",_n])
?(_n > 0):goto "loop"
```

This will loop 5 times.

---

[1] positions are arrays with the 3D coordinates - [easting,northing,height]. The `getpos` command queries an object or unit's position and returns this array.

Optimizing Scripts

Scripts are known to cause lag – so many people refrain from using them. But this is only half the truth: as long as you write your scripts correctly they won't have much impact on the game performance. There are 2 main reasons why scripts may cause lag:

- When a script is executed the engine works very fast through the script. Unless it encounters a wait ('~') or halt ('@') statement the engine tries to process the whole script in one go – fortunately BIS included an upper limit of commands to be processed within one simulation cycle. Else endless loops would be able to lock the game completely. This limit is about 100 instructions per cycle and script. Considering that some commands may well be quite CPU intense even that might cause the computer to slow down if many scripts are running at a time.

- When executing a script the game engine has to access the hard drive and load the script – even if the script has already been executed before. Usually it should reside in your file cache after being started for the first time, but don't bet on that if the mission has to load other stuff, too. If your hard drive is slow or very fragmented, or different scripts are started multiple times in a short time frame, then you might get some stuttering.

Both problems can easily be minimized by careful scripting. Using functions the game engine has to access the files only once (before the game starts), and using waits and halts it only has to check once every cycle whether it's time to continue – which is much less work than working trough a bunch of commands. Also file access can be spread out a bit by using short pauses – thus reducing the risk of lag even with scripts already residing in the file systems cache.

Taking a Break

Two commands enable us to temporarily stop the processing of a script:

`~<time>`	waits for a certain time (in seconds)
`@<condition>`	halts the execution of the script until a given condition is met.

Examples:	`~10`	*(waits for 10 seconds)*
	`@(player in (crew heli1))`	*(halts the script until the player is onboard the vehicle called heli1)*

Even the shortest wait will cause the script to stop execution and continue only after the check during the next frame has confirmed the time to be up. Just try using a wait of `~0.001` – unless your system is running at well over a thousand frames/sec this will always be much shorter than one simulation cycle. Still the game will stop the script execution and only continue during the next cycle.

Especially if you are using many loops or long scripts, a wait – even the shortest – will greatly improve the performance of your missions. If you have a very complicated condition or just no need to check your condition many times a second you could also use a small conditional loop with a short wait instead of the halt.

Example:

```
#loop
  ~1
?(alive leader team1):goto "loop"
```

This would check once a second if the team leader is still alive – something that surely doesn't need to be checked several dozen times every second.

Functions

Functions are like user defined commands, and like any OFP command they may return values. For example a function converting a unit's position to grid coordinates may return the grid location as a string, so it can be used to output the position on the screen or via radio messages. Functions are realized as *command strings* (see below), so that neither wait/halt nor `goto` instructions can be used. Another difference to normal scripts is that a script executing a function will always wait for the function to finish processing before continuing, while it wouldn't do so for a script. There's a big selection of ready-to-use functions at the OFPEC Editor's Depot (http://www.opfec.com), that you might want to check.

Using a Function

Functions, like scripts, are just simple text files. They usually carry the ".sqf" extension. They get preloaded (preferably in the init.sqs) using the `preprocessfile` instruction and are assigned a name:

```
FunctionName = preprocessfile "filename.sqf"
```

Example:

```
RandomSelect = Preprocessfile "randomselect.sqf"
```

This name can then be used to `call` the function whenever needed:

```
FunctionOutput = FunctionInput call FunctionName
```

Example:

```
_array = [unit1, unit2, unit3, unit4, unit5]
_selected_unit = _array call RandomSelect
```

The above example would use a function called *RandomSelect*[1] to select one of 5 entries from the given array at random, and then assign it to the local variable `_selected_unit`. We will use this function to explain their basic structure.

Structure of Functions

As mentioned before, a function is a command string. This signifies that:

- we can't use waits, halts
- we can't use the `goto` command
- we have to finish all commands with a colon (';').
- local variables are always shared with the script calling the function

The latter is the reason we have to declare private variables in the function. These will then be used like local variables in a normal script, i.e. a variable with the same name may exist in the script calling the function without causing any conflicts.

---

[1] *RandomSelect* is a function written by me, Daddldiddl, that selects a random entry from a given array. The function is available at my website or at the OFPEC Editor's Depot.

Example structure:

```
private [“_A”,“_B”,"_Result"];

_A=_this select 0;
_B=_this select 1;

_result = _A * _B;

_result
```

First we declare the private variables using the `private` instruction. Note that the variable names are declared as an array of strings, but used like normal local variables inside the function. Then we can assign the parameter values to the variables. This works the same as for scripts. After we have done our processing (the calculation) the result is assigned to a variable (`_result` in the example). This variable is then placed at the end of the script (note the missing ‘;’) and it’s value will be returned as the result of the function.

A more practical (but far from complicated) example is the *RandomSelect* function that has already been used as an example a few times:

```
private ["_ntotal","_random","_result","_target"];

_ntotal=count _this;
_random=random(_ntotal);
_result=_random - ((_random) mod (1));
if (_result==_ntotal) then {_result=_result-1};

_target=_this select _result;

_target
```

In this function there’s again the declaration of private variables at the beginning. Then entries of the array (`_this`), which is the parameter required by the function, are counted. A random number between 0 and the number of entries is drawn and the resulting *real number*[1] is converted to a full number by substracting the digits from the full number. Then, in the unlikely case that the random number actually was exactly the number of entries in our array, we have to substract 1 from the result. The `_result` variable is the used to select the respective entry from the array and the result assigned to the variable `_target`, which is then returned as the output of the function.

To select a random entry of any array the function now just has to be preprocessed once and can then be used as often as needed by using the `call` instruction.

---

[1] *real numbers* are non-integer numbers, that means they have digits (i.e. 1.01 or 1.00 are real numbers, while 1 or 2 are integers)

Multiplayer Scripting

Difficult Relations – the Server/Client Setup

In multiplayer missions you have a server and several clients (the players' computers). In some cases (non-dedicated) the server is a client at the same time, and in other cases (dedicated) the server is not. Server and clients are connected by LAN or Internet.

Time Spent Traveling

Information transmitted between the server and the clients needs a certain amount of time to reach its destination and get processed. This time is called *ping*, and is measured in milliseconds (1ms = 0.001 seconds). If you are connected via LAN, then your ping will be very low, sometimes so low, that the game will display a value of zero, meaning it takes messages less than a millisecond to travel in between server and the respective client. Over the Internet these times are usually much higher – 150ms and below is to be considered a good connection, and 300ms it is still playable – if lag occurs, ping may raise to much higher values temporarily. Note that this means that a message needs up to 0.3 seconds to travel from the client to the server, then has to be processed and finally the result distributed to all clients again.

That means that from the moment you do something on a client to the moment when all other clients know about this it may well take half a second and longer! And, it may easily take again as much time (i.e. over a second total) until you see the reaction to this (for example you shot someone playing on another client).

Everyone Look for Himself

If you start a script, it will be started on all computers playing the mission, unless you specifically prevent this. For example if you have a trigger that toggles as soon as there are no more enemies in a certain area, it will trigger on all clients and the server. If that trigger executes a script, then your script will be started on all these computers, too. This might be exactly what we want – for example when starting a cutscene. But in many cases this may lead to problems.

Imagine a script creating some enemy units - let's just say it creates one enemy squad. In singleplayer mode it would create that squad and we would be all happy. But in multiplayer it will result in as many squads as there are computers running the script, as each instance of the script would create one squad.

For the same reason random numbers created within a script will be different on each computer. If you do not synchronize the variables, and then try to use these numbers you might get some weird situations which are funny at best, but can ruin your mission and scripts at worst. Just imagine a script setting random weather/time for your mission. On one computer it might be bright sunshine, while on another it could be night time with a thunderstorm rolling over the hills…

Working in a Multiplayer environment

For the reasons mentioned above it is imperative that you plan in advance which scripts should run on which computers, what variables have to be synchronized between clients, and that you keep in mind that due to the transmission times, the machines will not be 100% synchronized if the mission is played over the Internet.

Querying the Server

To make a script run only on a certain server, we have to use a condition – executing the script only if the local computer fulfills this condition. The condition may be in the call starting the script, or in the first line of the script – exiting the script if it’s not the right machine. For example a script could run only on the server, or only on all clients. It could run only if a certain unit belongs to the local computer or if the local player is still alive, etc. What all these conditions basically test is whether a certain unit is local to the machine the script is running on.

Of course players are local to their respective machine, and all ai units (and game logics, vehicles, objects, etc.) usually are local to the server, as their simulation is done there. This is true until a player gets to be in command of an ai squad or is driving a vehicle. As soon as this is the case all simulation steps for these units/vehicles are done on the player’s machine. That’s the reason why you do not lag when flying or driving vehicles, or why your ai team reacts instantly to your commands. They are now local to your machine, not to the server! How can you test if a unit is local to the machine? You use the `local` query:

For example: `?(local unit1):”shell125” createvehicle (getpos unit1)`

This would create a 125mm tank shell at the units position and blow it to heaven - but only if the unit was local to the machine. This can be used to separate the server from clients. By putting a gamelogic on the map and naming it “server” (or whatever you like). As gamelogics are always local to the server, you can now easily query if the local machine is a client or the server.

Examples: `?(local server):...` *true if the local machine is the server*
`?not(local server):...` *true if the local machine is not the server*

Note: a non-dedicated server is always server and client at the same time. Using the above query ‘`?(local server):...`’ would result in the condition being *true*, as the gamelogic “server” is local to that machine. If you wanted something to execute on clients and non-dedicated servers (i.e. all machines controlled by players), but not on a dedicated server, you would have to use a little trick. In former examples we have seen the reserved variable `player` used a few times. This variable always references to the local player – who doesn’t exist on a dedicated server. That can be used to identify a dedicated server:

For example: `if (name player != "Error: No Vehicle") then...`

This condition will only be *false* on a dedicated server (i.e. *true* for all clients).

Synchronizing Variables over the Net

Sometimes it is necessary to have the same value for a variable on all machines. As scripts are running locally and all variables defined by these scripts exist only on the respective computer, the values have to be forced over the net using the `PublicVariable` instruction.

For example:
```
if (local server)
  Then {
    randomvalue = random (10);
    PublicVariable “randomvalue”;
  };
```

This would create a random value between 0 and 10 on the server, assign it to the variable `randomvalue`, and then distribute this variable to all clients. The value would be the same for all clients, and the variable could be queried by scripts on all machines.

General Hints

The following recommendations are not just scripting hints, but may make your life easier when preparing missions for OFP in general and make your scripts work properly in MP.

- If you want to name the group of players (e.g. '`team1=group this`'), put the instruction in the init lines of all playable units of that team. If you only put it in one unit's init line and that unit is not played (neither by a player nor ai), then the init line is not executed and hence no name is assigned.
  If the group has ai units you can just put it in the init line of one of the ai units and not worry about the playable units. That way you can be sure that the name is assigned to that team, so you can query it during the game:
  `?(("alive _x" count (units team1)) > 0):…`
- If you create a unit that wasn't on the map before during the game, the engine has to load all missing textures and geometries during the mission. Depending on your hardware this will cause more or less lag.
  When creating units make sure a unit of the same type was already placed in the mission editor. Even if you delete that pre-placed unit right at the mission start, the game will already have loaded the necessary 3d data and therefore it will not need to access the harddrive for that when you start creating the respective units ingame.
- Things created with `camcreate` only exist on the machine where they got created. In MP games you should use `createvehicle` or `createunit` instead. Be aware, that the effects will be shared with the other machines, even if you use `camcreate`. That means, a bomb camcreated on one machine will still kill other players, even if they don't see the explosion.

References

Websites

- www.flashpoint1985.com – the official website. There you find all patches, official add-ons, the official scripting reference and some helpful tools. The official forum that you can access from this site is also very good – especially the mission editing section is always worth a look!
- www.ofpec.com – whether you are looking for tutorials, ready-to-use scripts for common problems or anything else related to mission editing – go there and find it. They also have a forum where you should find answers to all your questions.
- www.theavonlady.org/theofpfaq - the one and only Operation Flashpoint FAQ (thou shall not have any... etc.). Not really an editing site, but with many useful links and answers to most OFP related questions.
- members.lycos.co.uk/bloodmixer/tutorial_editing_faq_114.html - web editing FAQ 1.14 by Bloodmixer.
- home.arcor.de/vektorboson/en/index.html - Vektorbosons scripting site. I recommend trying his console script, and browsing through his great tutorials!
- daddldiddle.de - my own little site. Apart from the missions I made, you will find some helpful scripts and of course always the newest version of this tutorial (look in the download section for “Tutorials”).

Tutorials/References

Sorry, no download links provided here– use Google to find the documents without. I got many of these a long time ago or sent to me by others. In some cases the original download sites do not exist anymore, but there should be still sites hosting the documents. Some of these can also be found in the download section of my own website (daddldiddl.de)

- **Johan Gustafson’s Scripting Tutorial 2nd Edition** – great scripting tutorial, with more complete examples but a bit different emphasizes. Also try his editing tutorial (both tutorials are available at OFPEC).
- **OFP ReadMe** (check your Operation Flashpoint: Cold War Crisis cd-rom) – the readme contains a short introduction to mission editing (not scripting) in English, German, French, Italian and Spanish. Good enough for making your first mission.
- **Lesters Inoffizielle Deutsche Operation Flashpoint Befehlsreferenz V4.52** *(in German)* – the best command reference ever! 16 pages packed with almost all scripting commands and short (but intuitive) explanations. Print it with 2 pages per sheet and you got a handy reference for a quick lookup anytime. Even includes a short weapons/object/animations/unit reference...
- **Mr. Murrays Operation Flashpoint Editing Anleitung 2.0** *(in German)* – excellent and extensive editing documentation (189 pages packed with information on editing, scripting and even basic add-on making). If you understand (or are) German, this is what you need to enter the world of OFP mission making!
- **The ultimate editing manual** – extensive html documentation by Rob, Hangfyre, RED and snYpir. More a collection of several short tutorials covering many different topics from using sound and music to making your own briefing and other stuff.
- **Unofficial Operation Flashpoint Command Reference Manual V1.03** – extensive but unfortunately not quite up-to-date reference (PDF, 99 pages). Contains all scripting commands up to OFP V1.3, object and animation lists, and more – unfortunately some descriptions are missing or incomplete. Still, a good reference to have around.