Master

BI STUDIO: BASICS OF SCRIPTING

INTRODUCTION

Scripting is the main way for:

- Creating new game functionality.
- Controlling timelines, events, dialogs, cut-scenes, etc.

NOTE:

- ***Scripts*** can be edited with a simply text editor (e.g. ***MS Notepad***).
- If special characters are used outside of the **ASCII**, the ***script*** may be encoded as **UTF-8**.
- All text editors have an ***Encoding*** option for saving a file.

NOTE: ***Game engines*** of the **BI Studio** are written with the **C++**. Therefore, the ***scripting language*** is based on the **C++**. Learning the **C** or **C++** is useful for understanding this.

Some principles of the ***scripting***:

- The purpose of the ***scripting*** is to run your game features that cannot be done otherwise.
- Players will use your game features.
- Your game features can be implemented with **scripting**.

NOTE: ***Scripts*** use system resources, therefore they affect hardware performance.

SOME DEFINITIONS

Signs:

- ***ampersand*** **(&)** used for a comparing operation **AND (&&)**
- ***semicolon*** **(;)** used for:
  - beginning a ***comment*** in the **SQS** syntax
  - ending a ***statement*** in the **SQF** syntax
- ***colon*** **(:)** used for:
  - the ***arithmetic*** "***if***" **(?:)** in the **SQS** syntax
  - a ***case*** label within the ***switch*** control structure
- ***hesh*** **(#)** used for a label in the **SQS** syntax
- ***exclamation*** **(!)** used for inversing a function result
- ***hyphen*** **(-)** used as the unary and binary minus
- ***tilde*** **(~)** used for absolute timing in the **SQS** syntax
- ***at*** **(@)** used for conditional timing in the **SQS** syntax

Editor Entities:

- A ***Unit*** is a manned entity controlled by either ***artificial intelligence*** **(AI)** or a player.
- A ***Vehicle*** is a mobile unmanned (*empty*) entity that can be controlled by both **AI** and a player.
- A ***Object*** is a static unmanned (*empty*) entity that can be controlled by both **AI** and a player

Common terms:

- ***Scripting*** is writing of ***scripts***.
- A ***script*** is a ***program*** piece functionally completed.
- A ***program*** is a set of instructions ordered by ***syntax***.
- ***Syntax*** is an order of writing ***program expressions***.

SYNTAX

The proprietary ***scripting syntaxes*** by the **BI Studio**:

- **SQS** is single-line based ***syntax***. This means a ***statement*** cannot span over multiple lines.
  NOTE: The **SQS** is introduced in the ***Operation Flashpoint*** **(OFP)** and in the **VBS1 (**from the **BI Sim**).
- **SQF** is multi-line based syntax. This means a ***statement*** can span over multiple lines.
  NOTE: The **SQF** is introduced in the ***Armed Assault*** **(ArmA 1)** and in the **VBS2 v.1x**.

NOTE: You can use both of these.

Constructs:

- A ***block*** is a kind of a ***statement*** as sequence of statements delimited by ***curly braces*** **{**}.
  NOTE: The empty ***block*** may be useful for expressing that nothing is to be done.
- A ***Control Structure*** (see below)
- A ***Expression*** (see below)
- An ***Identifier*** (see below)
- An ***Operand*** (see below)
- An ***Operator*** (see below)
- A ***Statement*** (see below)

SQS Features

Binding

Example	Description
STATEMENT1 STATEMENT2	The **SQS** statements are separated by ***line-breaks***. **NOTE:** The **SQS** statements can be separated in-line by ***comma*** **(,)**: STATEMENT1, STATEMENT2
{STATEMENT1, STATEMENT2}	The **SQS *block***-statement can be only placed on the single line.

Comment

SQS comment can begin with:

Semicolon	Command
; it is a **SQS**-style comment	**comment** "It is a **SQS**-style comment" **NOTE:** This was introduced in the ***Operation Flashpoint: Resistance*** v1.85.

Constructs

See the *Control Structures* below.

Special Structures

#Label

Example	Description
#Label1 CODE **goto** "Label1"	You can define a ***label*** and use the ***goto*** command to jump back to the ***label***.

Conditional Expression

Example	Description
? CONDITION**:** STATEMENT1, STATEMENT2	You can query a condition for executing the code. **NOTE:** The **"?:"** means an ***arithmetic if***.

Other SQS Features:

- The **SQS** syntax has the **goto** command.
- The **SQS** syntax cannot return a ***result*** of a called script back to the calling script.
- Due to the single-line-based **SQS** syntax, it is not possible to create multi-line string ***constants***. To overcome this, you have to store the ***constants*** in separate ***.sqf** files and load them using **loadFile** or **preprocessFile** command.
  NOTE: The **preprocessFile** command uses C-style preprocessor, this means you can use both the // and /* */ comments and also **#define** macros.

SQF Features

Binding

Example	Description
STATEMENT1; STATEMENT2;	**SQF** statements are separated by ***semicolon* (;)**. **NOTE:** The ***line-breaks*** cannot be used for ending the ***statements***.

{
STATEMENT1;
STATEMENT2;
}
	The **SQF *block***-statement can span through multiple lines.

Comment

Example	Description
comment "It is **SQF**-style comment";	A **SQF** comment begins with the ***comment*** command, and can span through multiple lines:

NOTE: You can use C-style comments:

// the line C-style comment	/*the block C-style comment*/

Constructs

See the *Control Structures* below.

Other SQF Features:

- The **SQF** syntax has the ***else*** statement.
- A ***statement*** can span through multiple lines if it is enclosed by ***brackets* []**.
- The **SQF** syntax can return a ***result*** of a called script back to the calling script.

SQS vs SQF

Comment

SQS	SQF
; a comment **comment** "a comment"	**comment** "a comment"; //in-line C-style comment /*block C-style comment*/

Waiting

SQS	SQF
@CONDITION	**waitUntil** {**if** (CONDITION) **exitWith** {***true***}; ***false***};

Timing

SQS	SQF
~TIME ; time in seconds	**sleep** TIME; // time in seconds

Single-Condition

SQS	SQF
? CONDITION**:** CODE	**if** (CONDITION) **then** {CODE};

Multi-Condition

SQS	SQF
?CONDITION**: goto** "SKIP" CODE2 **goto** "END" **#SKIP** CODE1 **#END**	**if** (CONDITION) **then** {CODE1} **else** {CODE2};

Structured Condition

SQS	SQF

? CONDITION1: **goto** "SKIP1"
? CONDITION2: **goto** "SKIP2"
 	
; DEFAULT
CODE3
goto "END"
 	
#SKIP1
CODE1
goto "END"
 	
#SKIP2
CODE2
goto "END"
 	
#END
	switch (VARIABLE) **do**
	{
	case CONDITION1: {CODE1};
	case CONDITION2: {CODE2};
	default {CODE3};
	};

While-Iteration

SQS	SQF
#ITERATION CODE **?**CONDITION: **goto** "ITERATION"	**while** {CONDITION} **do** {CODE};

Step-Iteration

SQS	SQF
_n = 0 **#ITERATION** CODE _n = _n + 1 **?** _n < COUNT: **goto** "ITERATION"	**for** [{_n = 0},{_n < COUNT},{_n = _n+1}] **do** {CODE}; Alternative syntax: **for** "_n" **from** 0 **to** COUNT **step** VALUE **do** {CODE};

Exiting

SQS	SQF
? CONDITION: **goto** "Exit" CODE2 **goto** "END" **#Exit** CODE1 **Exit** **#END**	**if** (CONDITION) **exitWith** {CODE1}; CODE2;

SCRIPT

A ***script*** is a functionally completed code that performs a specific task. It can be accessed as a ***function***.
The Function Types:

- A ***void function*** is used for process where a timing is important (i.e. controlling the actions).
- A ***value function*** is used for process where a result is important.

A ***function*** can accept ***parameters*** and return a ***result*** (a ***return value***) or a ***handle*** back to a caller.
One ***function*** can be shared by multiple callers.

Executing Commands

COMMAND: *exec*
Introduced in*:* *Operation Flashpoint* **(OFP) v1.00**
Compilation: A ***.sqs *script*** is compiled internally.
Execution: It executes a **SQS**-syntax script.
NOTE: Within the ***script***, the reserved local variable ***_time*** contains the time in seconds elapsed since the ***script*** started.
Alternative: The **execVM** with **SQF** syntax.
Syntax: [*arguments*] **exec** *script*
Parameters:

- ***arguments* (optional)**: *Any Value* passed into the script via the magic variable ***_this*** as an ***Array***.
- ***script:*** *String* - a name of the script.
  It should be given relatively to:
  - a mission folder ***\scripts***
  - a campaign subfolder ***\scripts***
  - a global folder ***\scripts***

  NOTE: It is searched there in the same order.

If this is referred to a ***script*** packed together with an addon, the path should be "\<addon>**script.sqs**".

NOTE: The <*addon*> means the name of the ***.pbo** file without extension.
Return: *Nothing*
Example:

```
[player, jeep] exec "getin.sqs"
```

NOTE:

- The ***exec*** starts a new thread for a called script, and does not wait for it to finish.
- The ***exec*** returns nothing from the called script back to a caller.

Example:

- Content of **VoidFnc.sqs**:

```
hint "Sentence1";
```

- Executing the **VoidFnc.sqs**:

```
[] exec "VoidFnc.sqs";
hint "Sentence2";
```

- Output would be:

```
Sentence2
Sentence1
```

COMMAND: ***call***
Introduced in: ***Operation Flashpoint: Resistance*** **(OFPR) v1.85**
Compilation:

- A ***.sqs** script is compiled internally.
- A ***.sqf** script must be precompiled via the ***expressions***:
  - **compile preProcessFile**
  - **compile preprocessFileLineNumbers**

Execution: It executes the **SQS**-/**SQF**-syntax ***script***.
Alternative: No
Syntax:
[*arguments*] **call** {*code*}
[*arguments*] **call** *variable* // the code is precompiled and saved into the ***variable*** else anywhere
Parameters:

- ***arguments*** **(optional):** *Any Value* passed into the ***script*** via the magic variable ***_this*** as an ***Array***.
- ***code***: *Code* or a called script returned via the commands **loadFile** (SQS) or **preProcessFile** (SQF).

Return: *Anything* - a last value from the called ***script***.
Examples:

SQS	SQF
_LastValue = [] **call** {"x = 3"} ; the code	_LastValue = [] **call compile** {"x = 3"}; // the code

_n = 3 ; the variable
_LastValue = [] **call** _n; the result is 3
	_n = 3; // the variable
	[] **call compile** _n; // the result is 3

_CalledScript = **loadFile** "script.sqs" _LastValue = [] **call** _CalledScript	_CalledScript = **compile preProcessFile** "script.sqf"; _LastValue = [] **call** _CalledScript;

NOTE:

- The ***call*** does not start a new thread for the called ***script***, and waits for it to finish.
- The ***call*** returns a last ***value*** from the called ***script*** back to the caller.

Example:

- Content of **ValueFnc.sqf**:

```
hint "Sentence1";
```

- Executing the **ValueFnc.sqf**:

SQS	SQF
_Sentence1 = [] **call loadFile** "ValueFnc.sqf" **hint** "Sentence2"	_Sentence1 = [] **call compile preProcessFile** "ValueFnc.sqf"; **hint** "Sentence2";

- Output would be:

```
Sentence1
Sentence2
```

COMMAND: *execVM*
Introduced in: *Armed Assault* (ArmA 1) v1.00
Compilation: A ***.sqf** script is compiled internally via the **preprocessFileLineNumbers** command.
Execution: It executes a **SQF**-syntax ***script***.
Alternative: The ***exec*** with **SQS** syntax.
Syntax: [*arguments*] **execVM** *script*
Parameters:

- ***arguments* (optional):** *Any Value* passed into the ***script*** via the magic variable ***_this*** as an ***Array***.
- ***script:*** *String* - a name of the script.
  It should be given relatively to:
  - a mission folder ***\scripts***
  - a campaign subfolder ***\scripts***
  - a global folder ***\scripts***
    NOTE: It is searched there in the same order.

If this is referred to a ***script*** packed together with an addon, the path should be "\<addon>\script.sqs".
NOTE: The <*addon*> means the name of the ***.pbo** file without extension.
Return:

- A *Handle* used to determine via the **scriptDone** command (ArmA 2) if the script has finished
- A *Boolean* value via the **isNull** command (ArmA 3) if the script has finished.
  NOTE: In ArmA 3, the ***handle*** is available within the ***script*** as the magic variable ***_thisScript***.

Example:

ArmA 2	ArmA 3

_handle = **execVM** "VoidFnc.sqf"; // the Boolean
waitUntil {
if (**scriptDone** _handle) **exitWith** {**true**};
false
};
	_handle = **execVM** " VoidFnc.sqf"; // the Boolean
	waitUntil {
	if (**isNull** _handle) **exitWith** {**true**};
	false
	};

NOTE:

- The ***execVM*** starts a new thread for a script, and does not wait for it to finish.
  NOTE: You can keep the program flaw until the called script finished (see above example).
- The ***execVM*** returns a ***handle*** from the called script back to the caller.

Example:

- Content of **VoidFnc.sqf**:

```
hint "Sentence1";
```

- Executing the **VoidFnc.sqf**:

```
_Sentence1 = [] execVM "VoidFnc.sqf";
hint "Sentence2";
```

- Output would be:

```
Sentence2
Sentence1
```

COMMAND: *spawn*
Introduced in: *Armed Assault* (ArmA 1) v1.00
Compilation: A ***.sqf** script must be precompiled via the ***expressions***:

- **compile preProcessFile**
- **compile preprocessFileLineNumbers**

Execution: It executes a **SQF**-syntax script.
Alternative: No
Syntax: [*arguments*] **spawn** {*code*}
Parameters:

- ***arguments* (optional):** *Any Value* passed into the ***script*** via the magic variable ***_this*** as an ***Array***.
- **code**: *Code*

Return:

- A *Handle* used to determine via the **scriptDone** command (ArmA 2) if the ***script*** has finished
- A *Boolean* value via the **isNull** command (ArmA 3) if the ***script*** has finished.
  NOTE: Since ArmA 3 v1.55, the ***handle*** is available within the script as the magic variable ***_thisScript***.

Example:

```
for "_i" from 0 to 100 do {_null = _i spawn {diag_log _this;}; // Result: 51, 1, 2...49, 50, 0, 52, 53...100};
```

NOTE:

- The ***spawn*** adds a script precompiled into a ***scheduler***, and does not wait for it to finish.
  NOTE: When this is run depends on how the ***game engine*** is busy and how the ***scheduler*** is filled up.
- The ***spawn*** returns a ***handle*** from the called script back to the ***scheduler***.

Example:

- Content of **VoidFnc.sqf**:

```
hint "Sentence1";
```

- Executing the **VoidFnc.sqf**:

```
_Sentence1 = [] spawn compile preprocessFile "VoidFnc.sqf";
hint "Sentence2";
```

- Output can be:

Sentence1 Sentence2	Sentence2 Sentence1

The **scriptDone** command can be used to check if the script completed:

```
_handle = [] spawn compile preprocessFile "VoidFnc.sqf";
waitUntil {
        if (scriptDone _handle) exitWith {true};
        false;
};
hint "Sentence2";
```

Output would be:

```
Sentence1
Sentence2
```

Processing Functions

Processing the ***value function*** **(VFnc)**:

1. It is loaded as a *String* from a file via either the ***loadFile*** (SQS) or ***preprocessFile*** (SQF) command.
2. It is precompiled via the ***compile*** (SQF) command.
3. It is executed via the ***call*** (SQS/SQF)command.

A ***value function*** is run within the existing ***thread*** that waits for the ***result*** from the ***function***. The ***value function*** suspends other processes until it has completed. This means the ***value functions*** have to run faster than ***void functions***, and the result of the ***value functions*** has to be immediate and unambiguous.

Examples:

SQS	SQF
Example 1: /* Load, compile and call the ***function*** from another ***script*** and then save the ***result*** of this into the ***variable****/ _result = **call loadFile** "VFnc.sqf"	Example 1: /* Load, compile and call the ***function*** from another ***script*** and then save the ***result*** of this into the ***variable****/ _result = **call compile preprocessFile** "VFnc.sqf";
Example 2: /* Load, compile and save the ***function*** as a *String* into the global ***variable*** anywhere*/ Fnc = **compile loadFile** "VFnc.sqf" /* Call the ***function*** from the global ***variable*** and then save the ***result*** of this into other one*/ _result = **call** VFnc	Example 2: /* Load, compile and save the ***function*** as a *String* into the global ***variable*** anywhere*/ Fnc = **compile preprocessFile** "VFnc.sqf"; /* Call the ***function*** from the global ***variable*** and then save the ***result*** of this into other one*/ _result = **call** VFnc;
Example 3: /* Define and save the in-line ***function*** into the local ***variable*** within the ***script****/ _VFnc = {CODE} /* Compile and call the ***function*** from the local ***variable*** and then save the ***result*** of this into other one*/ _result = **call** _VFnc	Example 3: /* Define, compile and save the in-line ***function*** as a *String* into the local ***variable*** within the ***script****/ _VFnc = **compile** {CODE}; /* Call the ***function*** from the local ***variable*** and then save the ***result*** of this into other one*/ _result = **call** _VFnc;

NOTE:

- You can use the special ***variables*** and commands in the ***value functions***.
- A ***value function*** will return the ***result*** of the last ***statement*** executed.
  NOTE: It does not matter whether the last ***statement*** is followed by a semicolon (;) or not.
- The ***result*** has to be saved into a ***variable*** to access it later.

In-line Function

An ***in-line function*** is that is defined and called within same ***script***.

Example:

SQS	SQF

_VFnc = {
  _val = **_this select** 0; // It is external parameter
  if (_val > 5) **then** {"bigger"} **else** {"smaller"};
};
_result = [4] **call** _VFnc; // "smaller"
	_VFnc = **compile** {
	  _val = **_this select** 0; // It is external parameter
	  if (_val > 5) **then** {"bigger"} **else** {"smaller"};
	};
	_result = [4] **call** _VFnc; // "smaller"

Processing a ***void function*** **(VdFnc)**:

1. It is loaded internally via ***exec*** (SQS) or ***execVM*** (SQF) command.
   NOTE: In case the ***spawn*** (SQF) command, the ***script*** has to be loaded via either of:
   - **preprocessFile**
   - **preprocessFileLineNumbers**
2. It is compiled internally via ***exec*** (SQS) or ***execVM*** (SQF) command.
   NOTE: In case the ***spawn*** (SQF) command, the ***script*** has to be compiled via the ***compile*** command.
3. It is executed via either of the ***exec*** (SQS) or ***execVM***/***spawn*** (SQF) command.
   NOTE: In case the ***spawn*** (SQF) command, the ***script*** has to be precompiled (see above).

A ***void function*** is run in a new ***thread***, and the existing ***thread*** does not wait for a ***handle*** from the ***function***. Thus the ***handle*** is not accessible to the existing ***thread***. This prevents large and CPU intensive code from seizing up the ***program flow***.
The ***void function*** using either the ***execVM*** (SQF) or ***spawn*** (SQF) command will return its ***handle*** used with ***scriptDone*** or ***terminate*** command.
NOTE: In case the ***exec*** (SQS) command, the ***void function*** returns nothing back to the caller.
Example:
Contents of the **max.sqf**:

```
_a = _this select 0; // the external parameter
_b = _this select 1; // the external parameter
if (_a > _b) then {_a} else {if (_a < _b) then {_b} else {hint "A max value does not exist."}};
```

Executing the **max.sqf**:

```
// Load, compile and call the void function from another script and then save the result of this into the variable
maxValue = [3, 5] spawn compile preprocessFile "max.sqf"; // "5"
```

Locations
If scripts are placed in the mission/game folder, no path has to be used:

```
Handle = [] execVM "script.sqf";
```

NOTE:

- If subfolders are used, either a relative path or an absolute path has to be used:

```
Handle = [] execVM "scripts\script.sqf"; // the relative path
Handle = [] execVM "D:\scripts\script.sqf"; // the absolute path
```

- If a relative path:
  - The ***scripts*** folder is accessible within both the mission editor and a compiled mission file.
  - The ***game engine*** will look for the ***scripts*** folder in the mission folder and then in the game folder. In case the game folder, this way do not work if a mission is moved to another computer as the files are not packed with.
- If an absolute path, the ***scripts*** folder can be located anywhere.
- In the **VBS**:
  - The double-backslash syntax is not supported:

```
\\LAN_PC1\DriveC\
```

  - The basic ***scripts*** folder can be used for the mission editor:

```
C:\Users\%user_name%\Documents\VBS2\scripts
```

    The location depends on the ***operating system*** and a game version. No path has to be provided to find the ***scripts*** folder located here. This is useful for tests while a mission development. If a mission exported is run outside of the mission editor, the ***scripts*** folder is not accessible.

The default search for ***scripts*** called with no path or with a relative path:

- **Mission Editor**: mission > (VBS: docs\scripts) > game root
- **Singleplayer Mode**: mission > (VBS: root\scripts) > game root
- **Multiplayer Mode**: mission > (VBS: docs\scripts) > game root

Scripts can be executed from:

- External files within a game root
- ***Initializations*** of the mission entities
- ***Event Handlers*** of the addon configurations

Statements

A ***statement*** is a construct for expressing a process.
Statement Types:

- ***Declaration*** (via initialization):

Manual Initialization	Default Initialization

_Var = “”; // the local variable
Var = “”; // the global variable
	private [“Var”]; // the local variable

- ***Expression***:
  - ***Assignment***
  - ***Input/Output***:
    - ***Command***
    - ***Control Structure***
  - ***Function Call***

Expressions

An ***expression*** is a code that returns a result ***value***.
Expression Types:

- ***Assignment*** is a redefining of a ***value***:

_Var = “”; // the initialization
_Var = 1; // the assignment
	private [“Var”]; // the initialization
	Var = 1; // the assignment

- ***Operation*** is an expression including an ***operator*** and its ***operands***:

```
a+b;
```

- ***Command*** is a ***function***, including its ***arguments***, e.g.: _array **select** 0;

```
_Array = [1, 2];
_Array select 0; // the item 0 has got the value of 1
```

- ***Control Structure*** is a conditional ***function***:

```
if (CONDITION) then {CODE1} else {CODE2}
```

- ***Function Call*** is a ***function***, calling a ***script***:

```
Handle = [parameters] execVM “script”
```

Variables

A ***variable*** is a named ***object*** used to store a data. Different entities (e.g. scripts, triggers, objects and add-ons) can read and modify data in the ***variables***.

Rules of naming:

- A name may consist of the **ASCII** text: characters (**a-z, A-Z**), numbers (**0-9**) and underscore (_)
- A global name must start with a character, not a digit: e.g., *GlobalVariable1*
- A local name must start with underscore: e.g. *_LocalVariable2*, *_2LocalVariable*

NOTE: In **VBS**, validity of a name dynamically created can be checked using the **isValidVarName** command.

An ***identifier*** is a name of a ***variable***:

- Capitalized: *GlobalVariable*; *_LocalVariable*
- Underscored: *global_variable*; *_local_variable*

Variables are available in certain ***namespaces*** (areas). This feature prevents the ***variables*** from conflicts.

Local variables are available within a ***script***. It has access to them, including ***functions*** called. The ***local variables*** cannot be used for the editor entities (units, triggers, waypoints, etc.), but they can be used in PreProcessor EXEC commands.

NOTE:

- Some ***local variables*** predefined by the ***game engine*** (e.g. **_this**, **_pos**) may be available.
- ***Global variables*** are available within a computer where they are defined.
- The ***global variables*** can be used for the editor entities.

Public variables are available within the network. A value of a ***global variable*** gets broadcasted over the network using the ***publicVariable*** command. After the call of this command on a ***server*** the ***variable*** will have the same ***value*** on all ***clients***.

NOTE: If the ***value*** of the ***public variable*** changed, it will have to be passed via this command again.

Defining

The ***game engine*** automatically defines the ***variables*** on their ***initialization***.

Querying the undefined (uninitialized) ***variables*** returns an undefined value (**nil**):

scalar bool array string 0xe0ffffef - error

NOTE: The ***isNil*** command can be used to check whether a ***variable*** has been defined yet.

Initializing:

Initializing via an assignment	Initializing via the ***private*** command*
_txt = ""; // It is initialized	**private** ["txt"]; // It is initialized

*it is recommended for ***functions*** to avoid changing a ***value*** of an argument in a calling ***function***

NOTE: Since **VBS** v3.7 this automatic inheritance can be overridden via the **privateAll** command.

Scope

A ***global variable*** initialized is accessible on the computer ***scope***.

A ***local variable*** initialized is accessible on the same and lower ***scopes*** in a ***script***. To read the ***variable*** assigned at a lower ***scope***, it must be initialized before in the ***scope*** it is supposed to be read later in:

_txt = ""; // It is initialized
if (**alive player**) **then** {_txt = "Hello";};
hint _txt; // "Hello"
	private ["txt"]; // It is initialized
	if (**alive player**) **then** {txt = "Hello";};
	hint txt; // "Hello"

NOTE: If a ***local variable*** is initialized in a lower ***scope***, it is not accessible to higher ones:

if (**alive player**) **then** {_txt = "Hello";}; // It is initialized at the lower ***scope***
hint _txt; // The variable is undefined at the higher ***scope***

Functions are considered to be on a lower ***scope***, and share the ***namespace*** of a ***variable***:

```
_fnc = {_i = 2;}; // the definition of the in-line function
_i = 0; // it will be overwritten
call _fnc; // the in-line function returns its last statement
hint format ["%1", _i]; // "2"
```

To prevent a ***function*** from overwriting ***variables***, the **private** command should be used to initialize all ***variables*** within the ***function***:

```
_fnc = {private ["_i"]; _i = 2;}; // the definition of the in-line function
_i = 0; // it will not be overwritten
call _fnc; // the in-line function returns its last statement
hint format ["%1", _i]; // "0"
```

The ***scope*** of index ***variables*** used in the ***for-do*** iteration depends on the syntax of the iteration.

- the ***newvar*** syntax:

```
for "_i" from 0 to 2 do {CODE}; // The new "_i" is initialized within the iteration
```

- the ***specvar*** syntax:

```
_i = 1; // it is initialized before the iteration
for [{_i=0},{_i<2},{_i=_i+1}] do {CODE};
```

```
// it is initialized within the iteration
for [{_i = 0},{_i < 2},{_i=_i + 1}] do {CODE};
```

Destroying

Variables initialized will take up a memory.

Local variables are automatically destroyed after their ***script*** finished.

Global variables have to be manually destroyed via the ***nil*** keyword:

```
GlobalVar = nil;
```

DATA TYPES

A type of a ***variable*** specifies a type of data that the ***variable*** can contain. It is defined by the ***value*** of the ***variable*** on an ***initialization***. The ***variable's*** type can be changed by redefining it with another type of data:

```
Var = ""; // the String data type
```

```
Var = 0; // the Number data type
```

Data Types

Array is a list of items which can be of any data type, including other ***Arrays***:

- In a configuration of an addon (**config.cpp**) and of a mission (**description.ext**): color [] = {1, 0, 0, 1}
- In ***scripts***: color = [1, 0, 0, 1]

Boolean is a logic data:

- In a configuration of an addon (**config.cpp**) and of a mission (**description.ext**): **0** (***false***) or **1** (***true***).
- In ***scripts***: ***false*** **(0)** or ***true*** **(1)**.

NOTE:

- It can be returned by ***commands*** (var = **alive** unit1) and an ***operations*** (**if** (**true**) **then**{CODE})
- It can be assigned to ***variables*** (var = ***false*** or var = ***true***)

Code is a script data that consist of ***commands*** and their ***parameters***, and can be placed in ***.sqf** and ***.sqs** files.

In turn, one of the ***commands*** gets passed other ones: **if** (CONDITION) **then** {CODE}.

NOTE: ***Literals*** are usually represented by text within ***curly braces*** {}. Any such code is precompiled.

Config is a ***handle*** that represents either of a ***config*** file or a ***class*** within it.
NOTE: Since **VBS v3.6** the syntax and behavior of ***config*** files have changed.

- Before, a ***config*** would return **configName/configProperty**, since v3.6 this is preceded by **bin\config.cpp**/.
- Before, a comparison with an empty ***config*** would return ***true***, since v3.6 it will return ***false***.

The examples below assume the "dummy" ***configs*** to be non-existent, and the **missionConfigFile** and **campaignConfigFile** to be empty:

Example 1:

(**configFile**>>"dummy1")==(**configFile**>>"dummy2") (**missionConfigFile** == **campaignConfigFile**)
Before v3.6: ***true***, since v3.6: ***false***

Example 2:

(**configFile**>>"CfgPatches">>"")==**inheritsFrom** (**configFile**>>"CfgPatches " **select** 0)
Before v3.6: ***true***, since v3.6: ***false***

Example 3:

str configFile
Before v3.6: "", since v3.6: "**bin\config.cpp**"

Example 4:

str (**configFile**>>"dummy") **str missionConfigFile**
Before v3.6: "", since v3.6: ""

Example 5:

str ((**configFile**>>"CfgPatches ") **select** 0)
Before v3.6: "CfgPatches/access", since v3.6: "bin\config.cpp/CfgPatches/access"

Display is a screen element (see https://resources.bisimulations.com/wiki/Display).
Control is a dialog object (see https://resources.bisimulations.com/wiki/VBS:_Displays#Controls).
Editor Object **is** an editor entity (see https://resources.bisimulations.com/wiki/Editor_Object).

Group
Any ***unit*** belongs to its own ***group***:

- If ***units*** are linked together, they belong to the same ***group***.
- If a ***unit*** is not linked to anyone, it belongs to its own ***group***.

The **AI** makes many decisions on an entire ***group***, not on a ***unit***: behavior, combat mode, and waypoints.
NOTE: Empty objects do not belong to a ***group***.

Location is like an extended type of a ***marker*** (introduced in the **ArmA 1 v1.08**):

- It has a name, a side, a position, an area, and an orientation.
- It has a non-scaling map representation (icon and/or text, depending on a ***class***).
- It requires a class definition to define basic properties.
  NOTE:
  - The ***classes*** are defined in **bin\config.bin\CfgLocationTypes**
  - It can be changed, using ***commands***.
- It can be attached to an object with all its relevant properties inherited from the ***object***.
- It is local in a multiplayer mode, that means its properties are not synchronized
- Existing locations are set in a ***.pew** file of a terrain. When the terrain is exported to ***.wrp** file, the **island_name.hpp** is also produced. This contains the location names used in the ***.pew** file.
  NOTE: This ***.hpp** can be added into the **config.cpp** file of the terrain, using a **#include** directive.

NOTE: The **config.cpp** file of the terrain cannot be changed by ***commands***, but it can be read.

Namespace is a ***container*** used to store ***variables*** over specific ***scopes***.
Variables set in one ***namespace*** are not available in others, so the same name of a ***variable*** can be used in different ***namespaces***.
Namespace Types:

- **missionNamespace**: Retains content while in the same mission, or upon mission restores.
  NOTE:
  - Content gets lost upon mission restarts or retries.
  - This is where ***global variables*** are stored.
- **uiNamespace**: Retains content while in game.
  NOTE: Switching missions or user profiles does not reset content.
- **parsingNamespace**: The same scope as **uiNamespace**.
- **profileNamespace**: Retains content for the current user profile, even after restarting game.

NetObject is a special type of an ***object*** used with winches and joints.
NetObjects, like regular ***Objects***, are serialized when the mission is saved, and when loaded re-reference the same *Winch Or Joint*.
NOTE: Only available in **VBS2 v1.34+**.

Number **(SCALAR)** is a real number:

- The largest positive value is: 3.4028235e38
- The largest negative value is: -3.4028235e38

In scripts, it is possible to generate a representation of an infinite positive or negative number which compares even larger or smaller than the above two floating point limits:

- Positive infinity 1e39 = "1.#INF"
- Negative infinity -1e39 = "-1.#INF"

Degree is a type, between 0 and 360, returned by ***commands*** like ***acos*** and ***asin***.
Radian is a type, returned by ***commands*** like ***rad*** and ***deg***.

Object is either of the in-game (here) or in-editor ***object*** (see *Editor Object*). This is a generic reference for a man, vehicle and building. It can be animated, and have the **AI** associated with it.
Commands can refer to generic types, as much as specific subtypes.

```
_pos = getPos player;
_pos = getPos _MyHouse;
```

In general, the ***commands*** accept parameters of *Object* when a subtype is listed, but the ***command*** might not make sense or might not work on all ***objects***.
Types:

- A ***unit*** is a manned (**AI**) ***object*** that is animated.
- A ***vehicle*** is an unmanned ***object*** that is animated.
- A ***building*** is an unmanned object that can be animated.

NOTE: A ***joint*** is a connection used to couple different ***objects*** together (e.g., **setDriveOrientation**).

Script is a ***handle*** of operations called by the **spawn**/**execVM** commands.
When a ***script*** is done, the ***handle*** will contain **<NULL-script>** and the **scriptDone** command will return ***true***.
The ***script*** can be terminated by using its ***handle*** with the **terminate** command.
If the ***game engine*** does not contain a null-***value***, this can be created by calling an empty ***function***:

```
_hdlNull = [0] execVM {};
```

This ***handle*** will return ***true*** with the **scriptDone**, and could then be used to populate an array, for example, so that any type-specific test would not fail:

```
_hdlNull = [0] execVM {};
_handles = [_hdlNull, _hdlNull, _hdlNull];
_sqlHdl1 = execVM "script.sqf";
_handles set [1,_sqlHdl1]; // Now the 2nd element contains a real handle
_done = {scriptDone _x} count _handles; // run a type-specific command
```

Side
Types:

- **West (BLUFOR)** is predefined ***variable*** for entities that have the western side assigned.
- **East (OPFOR)** is predefined ***variable*** for entities that have the eastern side assigned.
- **Resistance (Independent)** is predefined ***variable*** for entities that have the resistance side (Independent/Guerrilla) assigned.
- **Civilian (Civilian)** is predefined ***variable*** for entities that have the civilian side assigned: people, empty vehicles, objects and dead of any side.
- **Unknown** is predefined ***variable*** for entities that have no side assigned.
  NOTE: It seems to only apply to empty groups.
- **sideLogic (Game Logics)**

String
A string of the **ASCII** characters enclosed by:

- single quotes ('OFP') for the *Operation Flashpoint* series
- double-quotes ("ArmA") for other of the *ArmA* series

Structured Text
See *Structured Text*

Target is an ***object*** interested for another one. The ***targets*** are internally used by certain ***commands*** to keep track of specific entities.

Special Types are data, which are not really ones, as they do not describe any ***value***, e.g.:

- ***Any Value***: the ***variable*** may to have any data type (excluding magic ones).
- ***Anything***: the ***variable*** may to have any data type or nothing.
- ***Nothing***: the ***expression*** has no value. It cannot be assigned to a ***variable***. It exists because each ***expression*** needs to return a ***value*** and needs to have a ***type***.
- ***objNull***: A non-existing ***object***. This ***value*** is not equal to anything, including itself.

Syntactical Helper Types are used for syntactically richer ***expressions*** than unary/binary operators do, e.g.:

- ***If*** is used in the ***if-then*** construct.
- ***While*** is used in the ***while-do*** construct.
- ***Switch*** is used in the ***switch*** construct.
- ***For*** is used in the ***for-do*** Iteration.

OPERATORS

An ***operator*** is a command that provide either of a basic mathematical or logical operation.
Operator Types:

- A ***unary operator*** requires one ***operand***: *operator* [*operand1*]
- A ***binary operator*** requires two ***operands***: [*operand1*] *operator* [*operand2*]:

An ***operand*** is a ***value*** or an ***expression*** given to an ***operator***.
NOTE:

- The ***Assignment*** operator (=) assigns a ***value*** to a ***variable***: *variable* = *value*.
- There does not exist any other assignment operator like C++ one.

Arithmetic Operators:

- The arithmetic operators can evaluate the different values
- The operand types for: *Number, String,* and *Array*.
- The arithmetic operators return a ***value*** of the *Number* type.

Unary Arithmetic Operators

Operator	Name	Example
+	Copy (for *Arrays*)	+Array
-	Negation	-a

Binary Arithmetic Operators

Operator	Name	Example
+	Addition (for *Numbers*); Concatenation (for *Strings* or *Arrays*)	a + b
-	Subtraction (for *Numbers*)	a - b
*	Multiplication (for *Numbers*)	a * b
/	Division (for *Numbers*)	a / b
%; mod	Modulo (for *Numbers*)	a % b; a **mod** b
^	Raise to the power of (for *Numbers*)	a ^ b

Modulo returns the remainder of the division (see Math Commands).

Array Operations:

- ***Operands*** of an array operation must be a type of the *Array*.
- The array operation returns a ***value*** of the *Array* type.

Unary Array Operations

Operator	Name	Example
+	Copy	+Array

The *Array* can be assigned:

- By reference: if you assign ***array1*** to ***array2*** and change ***array1*** afterwards also ***array2*** is changed:

```
array1 = [1, 2];
array2 = array1;
array1 set [0, 5]; // array1 = [5, 2], and array2 = [5 ,2]
```

- By copy: if you assign ***array1*** to ***array2*** and change ***array1*** afterwards the ***array2*** is not changed:

```
array1 = [1, 2];
array2 = +array1;
array1 set [0, 5]; // array1 = [5, 2], but array2 = [1, 2]
```

Binary Array Operations

Operator	Name	Example
+	Concatenation	Array1 + Array2
-	Removal	Array1 - Array2

- ***plus*** (+) attaches second operand on the end of first one:

```
array1 = [1, “two”];
array2 = [3, “two”, 4];
array3 = array1 + array2; // array3 = [1, “two”, 3, “two”, 4]
```

- ***minus*** (-) extracts all elements of second operand from first one by a ***type*** and a ***value***:

```
array1 = [1, “two”, 3, “two”, 4];
array2 = [“two”, 3];
array3 = array1 - array2; // array3 = [1, 4]
```

String Operations:

- ***Operands*** of a string operation must be a type of the *String*.
- The string operation returns a ***value*** of the *String* type.

Binary String Operation

Operator	Name	Example
+	Concatenation	string1 + string2

- “+” attaches second operand on the end of first one:

```
string1 = "Hello, ";
string2 = "World!";
string3 = string1 + string2; // string3 = "Hello, World!"
```

NOTE: There are not the unary string operations.

Logical Operators:

- The logical operators evaluate the *Boolean* values.
- ***Operands*** of a logical operator must be a type of the *Boolean*.
- The logical operators return a ***value*** of the *Boolean* type.

Unary Logical Operators

Operator	Name	Example
!; not	Not	!a; **not** a

NOTE: This ***operator*** returns the inverse ***value***: if the ***false*** is, then it returns the ***true*** and vice versa.

Binary Logical Operators

Operator	Name	Example
&&; and	And	a && b; a **and** b
||; or	Or	a || b; a **or** b
<>; xor	Xor	a <> b; a **xor** b

- **AND** returns the ***true*** if both operands are the ***true***
- **OR** returns the ***true*** if one or both operands are the ***true***
- **XOR** returns the ***true*** if either of the operands is the ***true***

The **NOR** and **NAND** operators can be simulated by the basic operators.
Combined logical operators

Operator	Name	Example
-	OR	!(a || b)
-	NAND	!(a && b)

- **NOR** returns the ***true*** if both operands are the ***false***
- **NAND** returns the ***true*** if one or both operands are the ***false***

Comparison Operators:

- ***Comparison operators*** compare two ***values***.
- ***Operands*** of a comparison operator must be one of the types:
  Number, Side, String, Object, Group, Structured Text, Config, Display, and *Control*
- The comparison operator returns the ***Boolean*** value:
  true if the comparison matches; ***false*** if not.

Comparison Operators

Operator	Name	Example
==	Equal	a == b
!=	Not Equal	A != b
<	Less Than	a < b
>	Greater Than	a > b
<=	Less Or Equal	a <= b
>=	Greater Or Equal	a >= b

NOTE: You cannot compare ***Boolean*** values:

- Comparing a ***Boolean*** value with ***true*** is the same as the value itself: a == ***true*** the same as a == a
- Comparing a ***Boolean*** value with ***false*** is the same as the inverse value: a == ***false*** the same as a ==!a

CONTROL STRUCTURES

A ***control structure*** is a ***statement*** used to control a program flow under certain conditions.
NOTE: Here, the ***control structures*** have no handling compared to other ***statements***. This is different from most imperative programming languages (like C), where ***control structures*** are implemented in the grammar. The controlling done by them is implemented by accepting code as an ***argument***.

The complex ***control structures*** like the ***while-do*** are implemented using helper types, like the ***while*** type.

A ***block*** is a set of ***statements*** grouped together within ***curly braces*** **{ }**.
NOTE: The ***block*** can be standalone, executable with calling command, or belong to a ***control structure***.

If-Then **Structure**
The ***if-then*** structure defines code executed if a condition is ***true***:

```
if (CONDITION) then {CODE}
```

Else**-Alternative**
The ***else***-alternative defines code executed when the condition is ***false***:

```
if (CONDITION) then {CODE_TRUE;} else {CODE_FALSE;}
```

NOTE: The CODE_FALSE is executed when CONDITION is ***false***.

The ***if-then*** structure can also be used to assign conditional ***values*** to a ***variable***:

```
state = if (alive player) then {true} else {false}; // state has the true if player is alive, otherwise the false
```

Since the "***if***" is a ***statement*** itself, it can be nested:

```
if (CONDITION1) then {
        if (CONDITION2) then{CODE1} else {CODE2};
}
else {
        if (CONDITION3) then{CODE3} else {CODE4};
};
```

Switch-Do **Structure**

The ***switch-do*** structure defines a code executed depending on a conditional ***value***:

```
switch (CONDITION) do {case VALUE1: {CODE1}; case VALUE2: {CODE2}; default {CODE3};};
```

NOTE:

- The ***default*** block can be used to catch ***values*** that are not defined in the case ***definitions***.
- There is no a ***colon*** (:) after the ***default*** tag.

NOTE: The ***switch-do*** structure can also be used to assign conditional ***values*** to a ***variable***:

```
_color = switch (side player) do {case west: {"ColorBlue"}; case east: {"ColorRed"};};
```

Iterations

Iterations are used to execute the same code for specific or infinite number of times.

WaitUntil-**Iteration**

The ***waitUntil*** iteration repeats a code if a condition is ***false***:

```
waitUntil {CODE; if (CONDITION) exitWith {true;}; false;};
```

NOTE: Here, the CONDITION is a return ***value*** of the CODE executed.

The process:

1. Execute the code.
2. Evaluate the condition: ***false*** - go back to the execution; ***true*** - quit the iteration, and then destroy it.

NOTE:

- If a condition is ***true*** on the start of the iteration it will quit after single iteration.
- Since the condition is tested after the execution, the iteration runs one or more times.
- The iteration can be only used within scheduled environment. It will resume on the next step.

```
_count = 0;
waitUntil {_count = _count + 1; _count < 2}
```

While-**Iteration**

The ***while*** iteration repeats a code if a condition is ***true***.

```
while {CONDITION} do {CODE}
```

NOTE: In the ***while*** iteration, the ***curly braces*** **{}** are used for the condition instead of ***parentheses*** **()**.

The process:

1. Evaluate the condition: ***true*** - go on to the ***block***; ***false*** - skip the ***block***.
2. If ***true***, execute the code.
   NOTE: If the condition is ***false*** on the start of the iteration, the code will never be executed.
3. Go back on to the condition.

NOTE:

- Since the condition is tested before an execution, it executes zero or more times. It will resume on the next step.
- The iteration can run up to 10,000 iterations if started from within a non-scheduled environment.

```
_count = 0;
while {_count < 2} do {_count = _count + 1;}
```

***For*-Iteration**

The ***for***-iteration repeats a code for a specific number of times.

NOTE: This is blocking and atomic in execution. It will not terminate or be interrupted after a certain time or after a number of fixed iterations (unless there is unexpected failure of the condition). This differs from ***while***-iteration, and has the potential to freeze up a ***program flow***.

```
for [{BEGIN}, {CONDITION}, {STEP}] do {CODE}
```

Parameters**:**

- BEGIN is an initialization of a start value before the iteration starts; part of parent and global variable spaces (inherits parent's local ***variables*** and is able to write to global variable space).
- CONDITION is evaluated before iteration.
- STEP is a degree of increasing or decreasing (with a negative ***value***) the start value.

The process:

1. Initialize the start value
2. Compare the start value: ***true*** - go on to the ***block***; ***false*** - skip the ***block***.
3. If ***true***, execute the code.
   NOTE: If a condition is ***false*** on start of the iteration, the code will never be executed.
4. Increase or decrease the start value.
5. Go back on to the condition.

Example:

```
for [{_i=0}, {_i<10}, {_i=_i+1}] do {hint "_i";}; // Displaying the numbers from 0 to 9
```

There is an alternate syntax of the ***for-do*** iteration that improves the performance of the iteration:

```
for "VAR_INIT" from STAR_TVALUE to END_VALUE do {CODE}
```

NOTE:

- VAR_INIT is the ***variable*** used to count the iteration; not part of parent's variable space nor part of global variable space; variables will not persist outside this iteration.
- START_VALUE is a ***value*** assigned to the ***variable*** before the iteration starts.
- END_VALUE is a ***value*** until the counter is incremented/decremented.

The process:

1. Initialize the ***variable*** with START_VALUE
2. Compare the START_VALUE to the END_VALUE. If this is deferent to, the code is executed:
   - If START_VALUE is less than END_VALUE, the VAR_INIT is incremented by 1
   - If START_VALUE is greater than END_VALUE, the VAR_INIT is decremented by 1
3. Go back to compare.

Example:

```
for "_i" from 0 to 9 step 1 do {hint "_i";}; // Displaying the numbers 0 to 9
```

NOTE: The default step is 1, but you can set the value at will:

```
for "VAR_INIT" from START_VALUE to END_VALUE step STEP_VALUE do {CODE}
```

NOTE: The STEP_VALUE defines the degree by which the start ***variable*** is incremented/decremented.

***ForEach*-Iteration**

The ***forEach***-iteration repeats a code for every item of an ***array***:

```
{CODE} forEach ARRAY
```

NOTE: The ***block*** is executed as long as a number of the items.

NOTE: You may use the special variable ***_x*** within the ***block*** that references to the current item of the ***array***.

```
_array = [unit1, unit2, unit3];
{_x setDamage 1;} forEach _array;
```

NOTE: The special variable ***_forEachIndex*** can also be used instead of the ***_x***.

NOTE: You can nest the **forEach** iteration so that a ***_x*** value of an outer one is available within an inner one:

```
{
	_unit = _x; // it is assigned to a local variable
	{
		hint format ["Unit: %1, Weapons: %2", _unit, _x]
	} forEach (weapons _x);
} forEach allUnits;
```

NOTE: Each the iteration will be executed within one frame.

***Count*-Iteration**

It is possible to use the ***count*** command instead of the ***forEach***.

```
{CODE} count ARRAY
```

The ***block*** is executed as long as the **count** of the ARRAY.

The ***value*** returned by the command is a count of return ***values*** that equal to ***true***:

```
_array = [unit1, unit2, unit3];
{_x setDamage 1; false} count _array;
```

NOTE: You can nest the ***count*** iteration so that a ***_x*** value of an outer one is available within an inner one:

```
{
	_unit = _x;
	{
		hint format ["Unit: %1, Weapon: %2",_unit,_x]; false
	} count (weapons _x);
} count allUnits;
```

NOTE: Each the iteration will be executed within one frame.

NOTE: The ***control structures*** (with exception of **count** one) return a ***value*** of the last ***expression*** evaluated. Therefore, there must not be a ***semicolon* (;)** after the last ***expression***, otherwise *Nothing* is returned.

```
_ret = if (CONDITION) then {VALUE1} else {VALUE2}; // return VALUE1or VALUE2
```

COMMON ERRORS

A ***compiler*** can generate some error messages in a game.

***Generic Error in Expression*:** the data type that an operator is expecting does not match:

```
String = "String" + 42
```

Invalid Number in Expression*:** the ***statement is incomplete or malformed:

```
Number = 2 + 3 +
```

Type Something Expected Nothing*:** the ***statement is incomplete, malformed, or non-existent.

Uncompleted Statement	Malformed Statement	Non-Existent Statement
var= ;	1 = 2	1 + 2 * 2

Type String Expected Code*:** a syntax error contained in a ***block as a piece of another ***statement***.
NOTE: The error will be identified as the piece of the original ***statement***, not on the line where it occurs. For instance, if there is a syntax error in a ***then***-block or an ***else***-block of an ***if***-statement, the error will be identified in front of the ***then*** keyword or ***else*** keyword, respectively.

Unknown Operator*:** the ***game engine attempted to parse something as an ***operator***, but could not find the given symbol:

```
string = "Hello, " concatenation "World!";
```

There are several reasons why this might happen:

- If a ***script*** uses a new ***operator***, and is run on an old version of the ***game engine***.
- When executing a formatted string, where a ***variable*** inside the ***statement*** is undefined:

```
_var = ;
hint format ["a = %1", _var]; // a = scalar bool array string 0xfcfffef
```

The ***game engine*** interprets a ***scalar*** as an uninitialized ***variable***, and the ***parser*** expects an ***operator*** as the next ***token***. The ***bool*** cannot be found in the list of operators (since it is not one).

scalar bool array string 0xe0ffffef*:** the ***variable does not exist.

```
if (format ["%1", _var] == "scalar bool array string 0xe0ffffef") then [{"undefined"},{"defined"}];
```

NOTE: The ***parser*** can point you to a line of correct code as an error, but the actual error is beneath that.

```
for "_i" from 0 to 1 do {
    _str = format ["string"
}; // The error message "Type String Expected Code"
```

In the example above, the error will be shown to the left of the ***do*** keyword (#do), but this is caused further.
NOTE: This applies to the **SQF** syntax, not to the **SQS**.

ADJUSTING (DEBUGGING)

Debugging is finding and fixing errors in a program.

Error Types:

- ***compile-time errors*** found by the *compiler*
- ***link-time errors*** found by the *linker*
- ***run-time errors* (*logic errors*)** found while the program is run.

NOTE: Generally, ***compile-time errors*** are easier to find and fix than ***link-time errors***, and ***link-time errors*** are often easier to find and fix than ***run-time errors* (*logic errors*)**.

To see error messages you can use:

- the ***.RPT** file (introduced in the *Armed Assault* v1.00): e.g., C:\Documents and Settings\organizer\Local Settings\Application Data\ArmA**arma.RPT**
  To use the file create a shortcut of it on the *Desktop*.
- the -**showscripterrors** switch (introduced in the *Armed Assault* v1.00).

To set the switch:

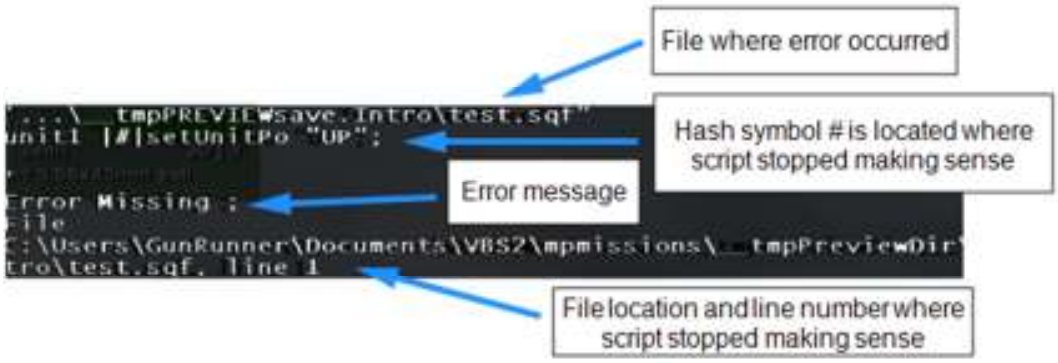
1. Create the game shortcut on the *Desktop*.
2. One-click RMB on the shortcut.
3. Choose ***Properties*** item from the context menu.
4. On ***Tab*** label, in the ***Object*** field, add the switch **–showscripterrors**:

```
"C:\Program Files\Bohemia Interactive\ArmA 2\arma2.exe" -showscripterrors
```

5. Click the ***Apply*** button.
6. Click the ***OK*** button.

Error messages appear at the top of the screen when the ***game engine*** loads the line of code but is unable to interpret this. The message displays the basic information about the error occurred.

NOTE: Actual error may be on another line.



Release: 24.05.2016
Update: 07.05.2017
Publisher: http://vied-arma.ucoz.com/