Achilles Documentation

Release 2.0.0

The Achilles Dev Team

Jun 29, 2019

First Steps

Warning: The current documentation is still a work-in-progress and does not reflect the current release version of Achilles.

Achilles is a gameplay modification for Arma 3. It expands the Zeus real-time editor with many new additions as well as provides bug fixes.

Achilles started as an expansion to Ares mod, which was created by Anton Struyk. Achilles became the *de facto* successor to Ares at the point the latter was no longer updated. Achilles has already grown into a *splendid* project, but new additions are still to come!

Tons of new modules Achilles provides new and exciting modules to spice up your gameplay and provide new experiences for all players.

Lots of customizability Using the powerful CBA settings framework, we provide stunning amounts of customizability for the Zeus interface to tailor it to just you.

Tried and tested As Achilles grew from Ares in 2016, it has accumulated over 250 000 unique downloads and come to a more and more polished product that has received many transformations over the years to make it even more splendid.

Open source Achilles is open source and has been so from the earliest of days. We are actively supporting the introduction of new contributions.

CHAPTER 1

First steps

Are you looking to create a new world of experiences for your multiplayer missions or just looking to have fun? Learn how to install Achilles and get playing with the revamped Zeus.

- **Getting started**: *Installing Achilles* | *Preparing a mission with Achilles*

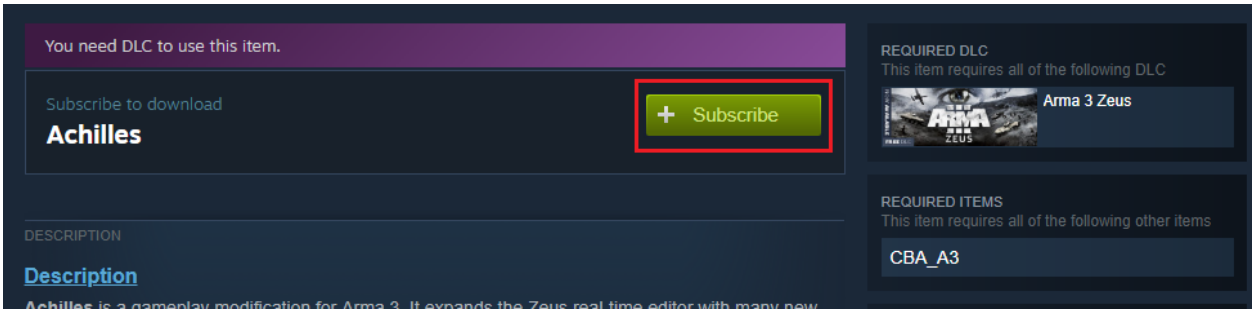
1.1 Installing Achilles

Contents

1.1.1 1. Installing from Steam Workshop

Installing the Achilles mod from its Steam Workshop page is a fairly easy task. You will need to install two mods, Achilles itself, and its dependency, CBA_A3.

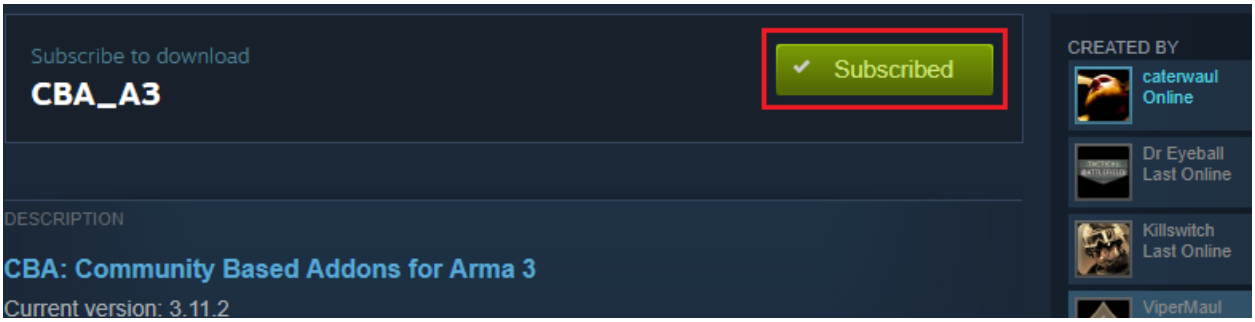
1.1. Subscribe to Achilles on Steam



This will download and automatically install the mod for you. Download the mod off Steam here.

1.2. Subscribe to CBA_A3 if prompted

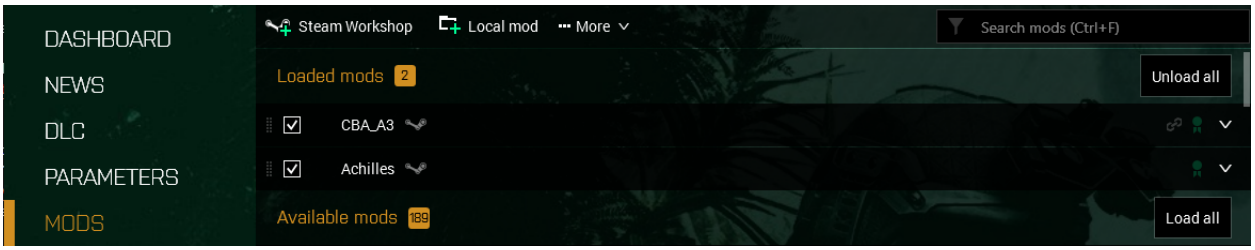
Ensure you download CBA_A3 from the Steam Workshop as well.



Make sure you subscribe to CBA_A3, or Achilles won't work properly!

1.3. Running the mods

Since Steam handled the installation automatically, you should now be able to see the new mods in your **ArmA 3 Launcher** when you start it, and be able to load them, as so.



You can now play the game and enjoy your new Zeus experience. Struggling to figure out how to get in and really experience Zeus? See our mission setup guide.

Preparing a mission with Achilles.

1.1.2 2. Installing from GitHub

Installing Achilles from GitHub is a slightly more complicated process. If you are not used to GitHub, or prefer an easier installation, we recommend you use the Steam installation guide. If you would rather download the mod from

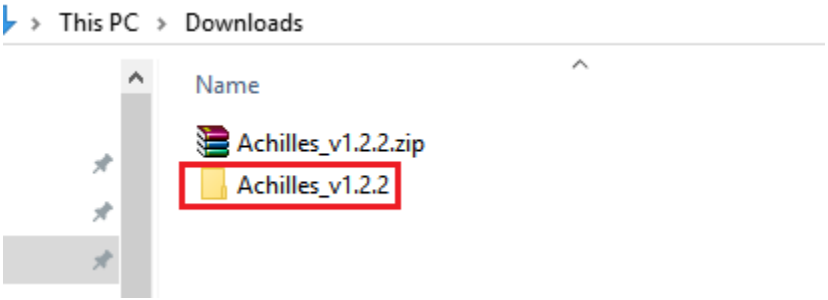
GitHub from whatever reason however, please read on.

2.1. Open the Achilles Releases on GitHub

You can find all current and old releases of Achilles.

Go ahead and download the latest release. If, for some reason, you want an older version, you can do that too, just download the zip.

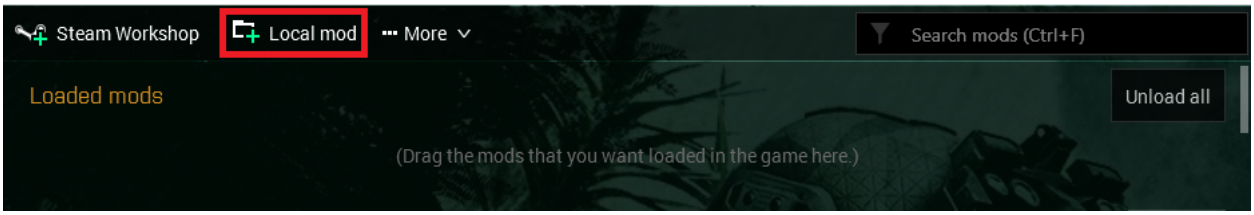
2.2. Extract the mod .zip



Extract the mod, so you get a folder like so. Open the folder, and you will find an @Achilles folder.

ArmA 3's launcher is configured to load local mods as well, as you might have noticed. Place the **@Achilles** folder somewhere safe. Most people recommend inside your ArmA 3 installation, but anywhere is fine.

2.3. Add the mod folder as a local mod



Click this option, navigate to where you placed the @Achilles folder, and voila, you added the mod into your launcher!

> **Warning:** Remember that if you will need to download CBA_A3, Achilles' dependency, a similar way to ensure the mod works!

Download CBA_A3 off their GitHub.

CBATeam's Releases

Thanks for reading, and have fun Zeusing!

1.2 Mission Setup

1.3 Frequently Asked Questions

1.4 Features

1.5 Support

1.6 Attribute Windows

1.7 Customizations

1.8 Dynamic Dialog

Contents

A good module needs a GUI. For this purpose we provide a simple to use, but powerful dynamic dialog system.

1.8.1 1. Creating the dialog

All dynamic dialogs are created using the `achilles_dialog_fnc_create` function.

Here's an example dialog:

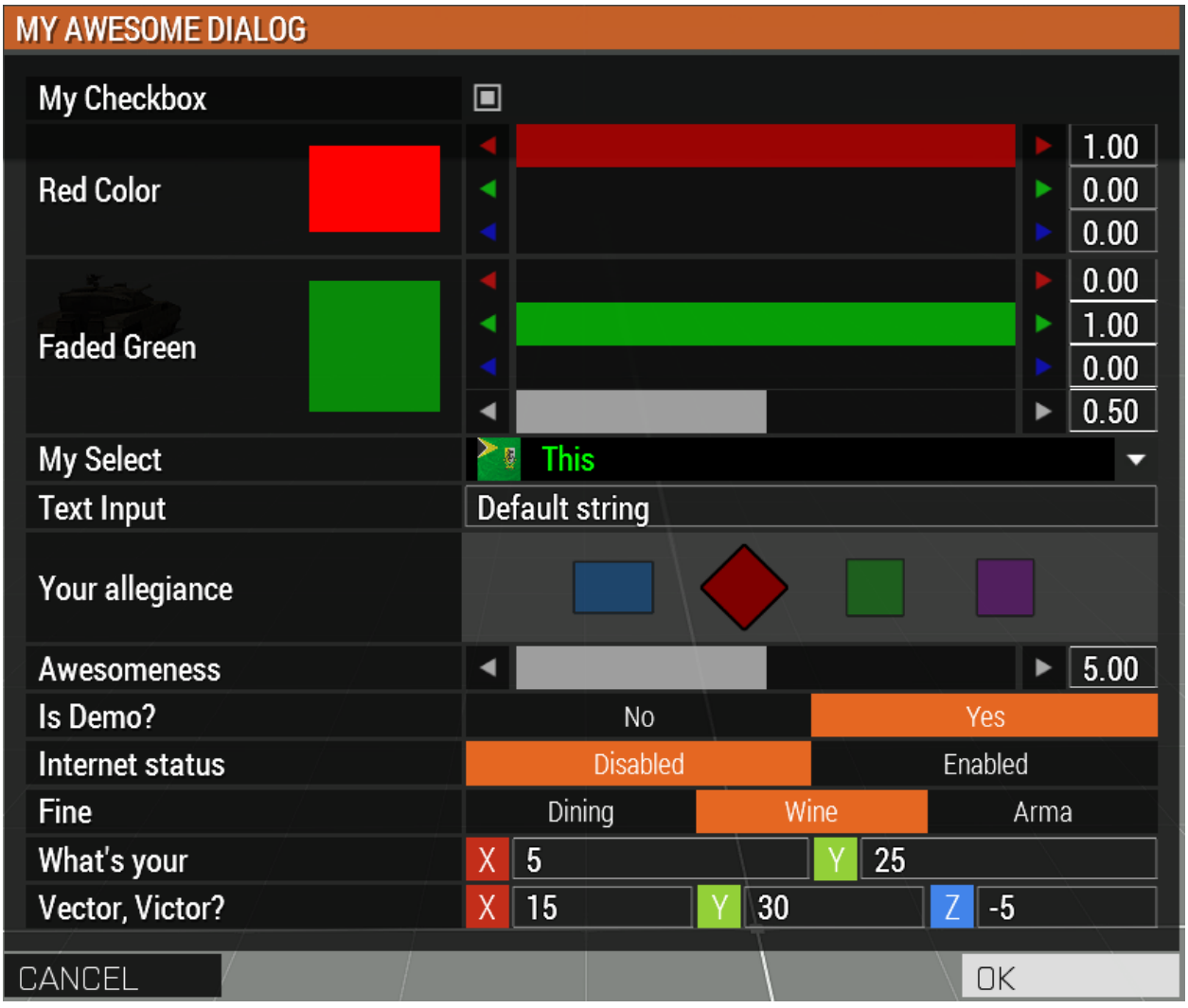
```
[
    "My awesome dialog",
    [
        ["CHECKBOX", "My Checkbox", true],
        ["COLOR", "Red Color", [1, 0, 0]],
        ["COLOR", "Faded Green", [0, 1, 0, 0.5]],
        ["SELECT", ["My Select", "With tooltip!"], [
            ["this", "my", true, "values"],
            [
                ["My basic thing"],
                ["This", "fancy tooltip", "\A3\Data_F\Flags\Flag_AAF_CO.paa", [0, 1, 0, 1]],
                ["Something else"],
                ["Other values"]
            ],
            1
        ]],
        ["TEXT", "Text Input", ["Default string", {params ["_notSanitized"]; _notSanitized}]],
        ["SIDES", "Your allegiance", east],
        ["SLIDER", "Awesomeness", [0, 10, 5, 2]],
        ["BLOCK:YESNO", "Is Demo?", true],
        ["BLOCK:ENABLED", "Internet status"],
        ["BLOCK", "Fine", [1, ["Dining", "Wine", "Arma"]]],
        ["VECTOR", "What's your", [5, 25]],
        ["VECTOR", "Vector, Victor?", [15, 30, -5]]
    ],
    {
```

(continues on next page)

(continued from previous page)

```
        // On success
        systemChat str _this;
        diag_log _this;
    },
    {
        // On cancel
        systemChat str _this;
        diag_log _this;
    }
] call achilles_dialog_fnc_create;
```

Produces the following dialog:



1.8.2 2. Function Arguments

The arguments for the actual dialog function is pretty simple, however, it can scale up to suit most of your needs.

Name	Type	Default
Dialog title	`STRING`	Required
Controls	`ARRAY`	Required
On Confirm	`CODE`	Required
On Cancel	`CODE`	`{}`
Arguments	`ANY`	`[]`

1.8.3 3. Control Arguments

Currently, there are 8 different controls for the dynamic dialog.

Name	Control Type	Alternative Control Types
Checkbox	`CHECKBOX`	N/A
Color (RGB or RGBA)	`COLOR`	N/A
Select dropdown	`SELECT`	N/A
Text	`TEXT`	N/A
Side selection	`SIDES`	`SIDES:ALL`
Slider	`SLIDER`	N/A
Block selection	`BLOCK`	`BLOCK:YESNO`, `BLOCK:ENABLED`
Vector (2 or 3 axis)	`VECTOR`	N/A
Description	`DESCRIPTION`	N/A

3.1. Checkbox control

The checkbox control is simple to use and doesn't have a lot of options.

Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"CHECKBOX"`	Display a checkbox type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Is checked?	`BOOL`	`BOOL`	Should the checkbox be checked?	`false`
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

Example:

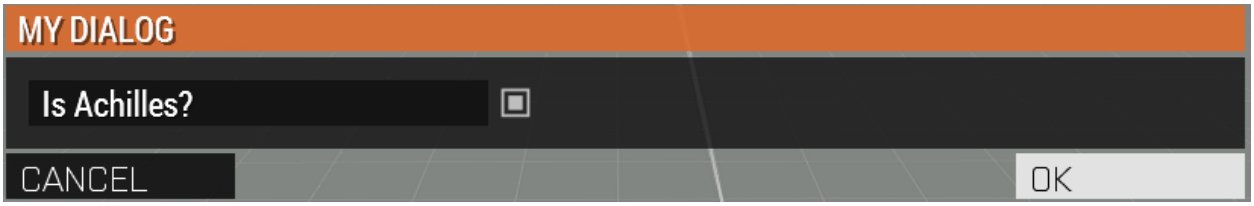
```
["My Dialog", [
    [
        "CHECKBOX",
        "Is Achilles?",
        true
```

(continues on next page)

(continued from previous page)

```
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:



3.2. Color control

The color control supports two different types. RGB *(red-green-blue)* or RGBA *(red-green-blue-alpha)*

There is no specific flag to set. The dynamic dialog system will automatically set the type depending on the value data array length.

Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"COLOR"`	Display a color type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Default color data	`ARRAY`	`[1, 1, 1]` or `[1, 1, 1, 1]`	What should the default color data be? If 4 arguments provided in the array, then it displays an RGBA control.	`[1, 1, 1]`
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

3.2.1. Color RGB

Example:

```
["My Dialog", [
    [
        "COLOR",
        "Blue color",
        [0, 0, 1]
    ]
], {}] call achilles_dialog_fnc_create;
```

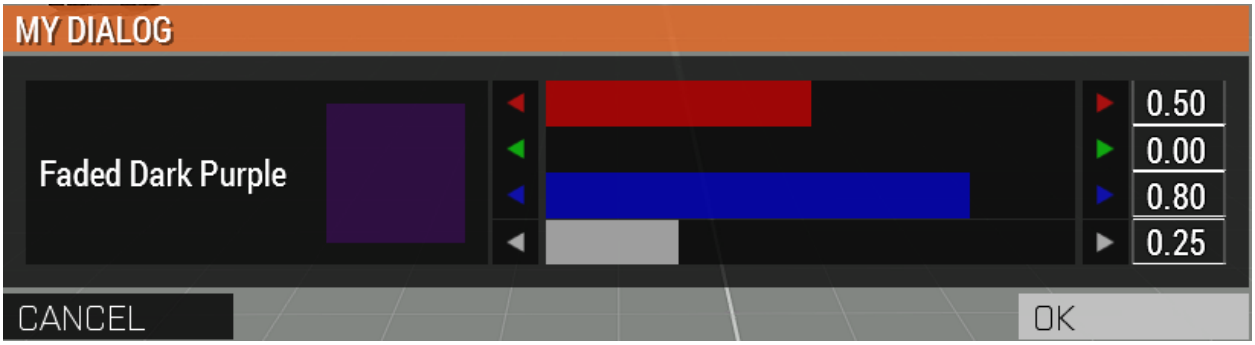
Result:

3.2.2. Color RGBA

Example:

```
["My Dialog", [
    [
        "COLOR",
        "Faded Dark Purple",
        [0.5, 0, 0.8, 0.25]
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:



3.3. Select dropdown control

Select dropdown is a dropdown list control that is very powerful. It allows for you to set tooltips, images, text colors, etc.

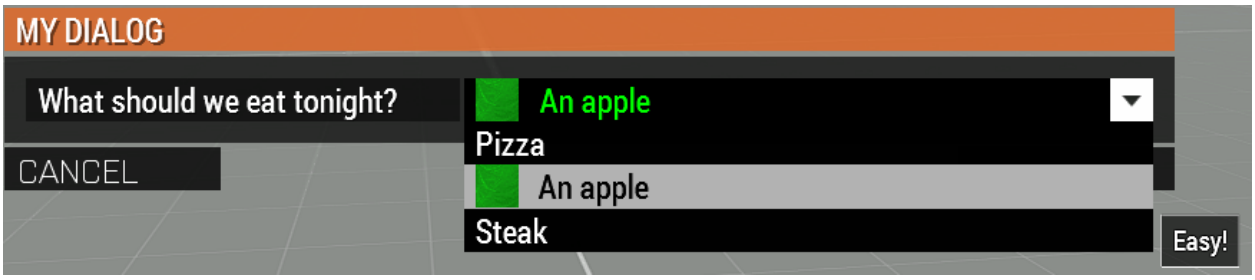
Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"SELECT"`	Display a select type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Array of selectable items	`ARRAY`	See "*3.3.1. Allowed values*"	Array of selectable elements that will be displayed to the user.	Required
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

Example:

```
["My Dialog", [
    [
        "SELECT",
        [
            "What should we eat tonight?",
            "Pick something delicious!"
        ],
        [
            [
                ["Flour", "Cheese", "Magic"], "Find it!", false
            ],
            [
                ["Pizza", "Delicious?"],
                ["An apple", "Easy!", "\A3\Data_F\Flags\Flag_green_CO.paa", [0, 1, 0, 1]],
                ["Steak"]
            ],
            1
        ]
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:



3.3.1. Allowed values

Name	Type	Default	Description
Value array of anything	`ARRAY`	Required	Once the user selects an item from the dialog and closes it (OK or Cancel) the selected value will be returned from this array.
Array of display values	`ARRAY` (See *display arguments*)	Required	An array of values that will be displayed to the user.
Default selected value	`SCALAR`	`0`	Allows to select which element will be the default selected one.

Display text arguments:

Below is a table with arguments for the display content of one element.

Name	Type	Default	Description
Display Name	`STRING`	Required	Dropdown item name to be displayed to the user.
Tooltip Name	`STRING`	`""`	Tooltip to display when the user moves his mouse over the dropdown item.
Picture Path	`STRING`	`""`	Path to the image to be displayed to the left of the display name.
Text Color	`ARRAY`	`[1, 1, 1, 1]`	The text color for that one dropdown item. **Requires color RGBA**.

3.4. Text control

The text control is a simple text box that allows users to input data into the box.

Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"TEXT"`	Display a text type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Default string to display	`STRING` or `ARRAY`	`STRING` or `["Default Text", {_this}]`	The default text what should be displayed when the control is first displayed.	Required
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

Default text has two options:

- Any string.
- Array of default text to display and the sanitize function or code to call.

This sanitize function receives the text the user is currently entering in `_this` variable. This function is called on each key press in the unscheduled enviornment.

> **Warning:** As this function is called on each key press, it has to be very quick.

Example:

```
["My Dialog", [
    [
        "TEXT",
        "What's the year?",
        "20"
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:

3.5. Side select control

A simple side selector control which allows the user to select between the 4 main sides.

- BLUFOR
- OPFOR
- Independent
- Civilian

An optional 5th side can be added: Logic side (`sideLogic`). This is achieved using the secondary control `SIDES:ALL`.

> **Warning:** It's highly recommended to provide a default value for the side. If not done so, then if the user doesn't select anything when prompted to, will result in a `nil` value in the dialog result.

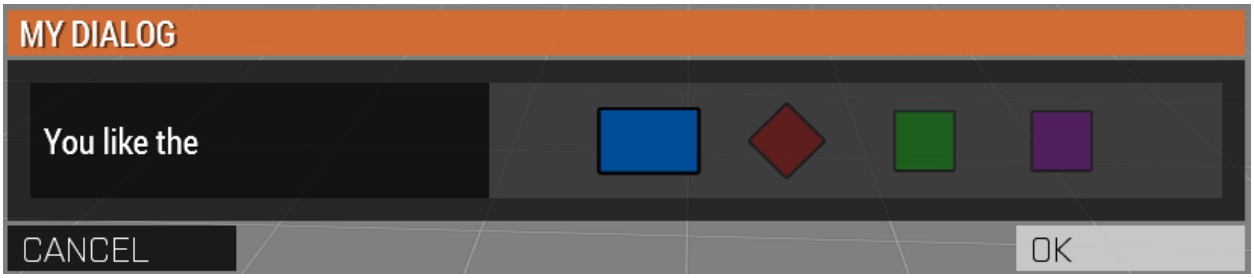
Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"SIDES"` or `"SIDES:ALL"`	Display a side type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Default side to show as selected	`SIDE`	`SIDE`	The default side that should be selected.	`nil`
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

Example:

```
["My Dialog", [
    [
        "SIDES",
        "You like the",
        west
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:



3.6. Slider control

The slider control is a simple slider that allows you to select a value in the defined range.

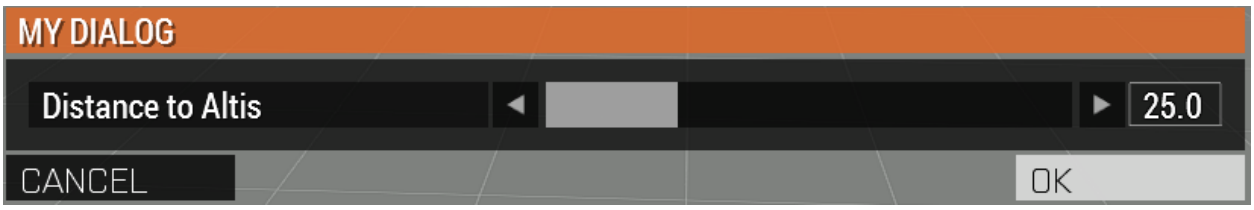
Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"SLIDER"`	Display a slider type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Array of slider settings	`ARRAY`	`[min, max, default, decimals]`	Array of the minimum and maximum allowed values of the slider, the default value to set the slider at and the decimal point.	`[0, 1, 0, 2]`
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

Example:

```
["My Dialog", [
    [
        "SLIDER",
        "Distance to Altis",
        [
            0,
            100,
            25,
            1
        ]
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:



3.7. Block selection control

The block selection is a way to select something without having to go into a select dropdown or something that the simple checkbox can't handle.

Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"BLOCK"`, `BLOCK:YESNO`, `BLOCK:ENABLED`	Display a block select type control. Allows to quickly use Yes/No or Enabled/Disabled type questions.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Array of block questions	`ARRAY`	See "*3.7.1. Allowed values*"	An array of data to be displayed to the user (not required if using the `:YESNO` or `:ENABLED` secondary controls.)	`[0, 1, 0, 2]`
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

3.7.1. Allowed values

Note: If using any of the secondary control types, then you do not have to add the questions.

Note: The maximum amount of items to select in the block control that can be added is 5.

To select the default value you can use the indexes of the question (0, 1, etc.) but if you only have 2 questions, then you can use a boolean.

If you are using the secondary control then you can also specify which control should be the default selected one. You can use a boolean to select the default question. `false` would be on the left and `true` would be on the right.

Examples:

```
["My Dialog", [
    [
        "BLOCK:YESNO",
        "Taras Kul",
        [true]
    ]
], {}] call achilles_dialog_fnc_create;
```

```
["My Dialog", [
    [
        "BLOCK",
        "She's",
        [
            2,
            [
                "Old",
                "Cool",
                "On Fire",
```

(continues on next page)

(continued from previous page)

```
                "Boring",
                "Other"
            ]
        ]
    ]
], {}] call achilles_dialog_fnc_create;
```

Results:

3.8. Vector control

The vector control works very similarly to the *color control*. As in it's dependent on the number of elements provided to display the number of axes you want.

If you provide 2 elements then you will only see the option to enter the *X* and *Y* axes, but if you provide 3 then the *Z* axis is added too.

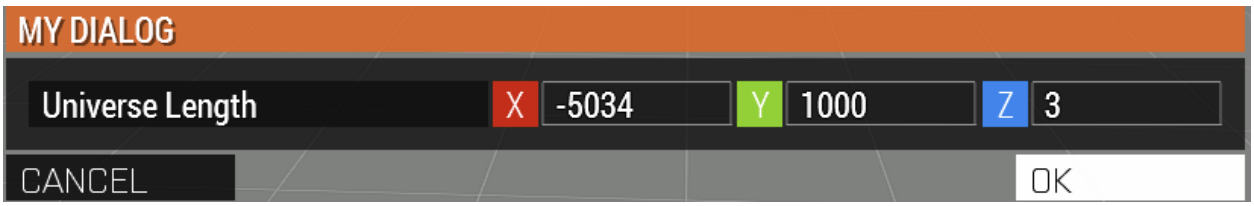
Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"VECTOR"`	Display a vector type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Array of vector axes	`ARRAY`	`[0, 0]` or `[0, 0, 0]`	The number of elements dictates if the *Z* axis should also be displayed.	`[0, 0]`
Force default value?	`BOOL`	`BOOL`	Should the given default value be forced? Should we ignore the last saved value?	`false`
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

Example:

```
["My Dialog", [
    [
        "VECTOR",
        "Universe Length",
        [-5034, 1000, 3]
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:



3.9. Description control

The description control is designed to display a multi-line text message to the user to describe anything you like.

If you want to display a multi-line message then you have to append the new line character (`\n`) to your string of text.

Note: This control does **not** return it's value when cancelling or confirming the dialog.

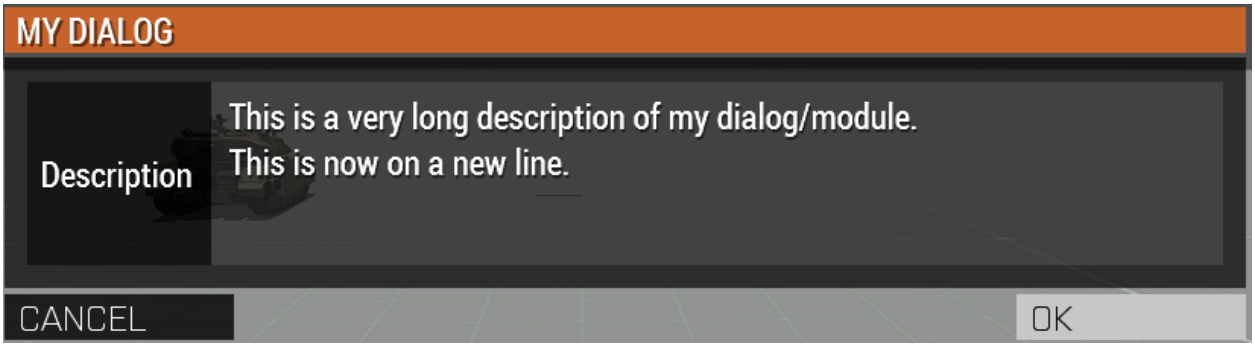
Arguments:

Name	Type	Allowed Values	Description	Default
Control	`STRING`	`"DESCRIPTION"`	Display a description type control.	Required
Display Name	`STRING` or `ARRAY`	`STRING` or `["Display Name", "Tooltip"]`	What does the control represent?	Required
Text to display	`STRING`	`STRING`	This text will be displayed to the user. To add a new line use the `\n` character.	Required
Resource function	`CODE`	`CODE`	Arguments are `[Control Group, Row Index, Default Value, Row Settings]`.	`{}`

Example:

```
["My Dialog", [
    [
        "DESCRIPTION",
        "Description",
        "This is a very long description of my dialog/module.\nThis is now on a new line."
    ]
], {}] call achilles_dialog_fnc_create;
```

Result:



1.8.4 4. On Confirm and On Cancel

On confirm and on cancel are two different scripts that will be executed depending on the following conditions:

- If the user presses the OK or Cancel buttons.
- If the user presses the Escape key.

When these scripts are called, data is passed in the `_this` variable.

Name	Type	Default
Array of selected values	`ARRAY`	N/A
Array of arguments (provided when calling the function)	`ARRAY`	`[]`

1.9 Hotkeys

1.10 Modules

1.11 Zeus Interface Changes

1.12 Waypoints

1.13 Architecture

1.14 Changelog

1.15 Coding Guidelines

Note: The coding guidelines are adopted from ACE3 and Achilles follows ACE3 standards. However, this page should be used for actual code guidelines not ACE3.

Contents

1.15.1 1. Naming conventions

1.1. Variable names

1.1.1. Global variable naming

All global variables must start with the Achilles prefix followed by the component, separated by underscores. Global variables may not contain the `fnc_` prefix if the value is not callable code.

Example: `achilles_component_myVariableName`

For Achilles, this is done automatically through the usage of the `GVAR` *macro family.*

1.1.2. Private variable naming

To make the code as readable as possible, try to use self-explanatory variable names and avoid using single character variable names.

Example: `_velocity` instead of `_v`

1.1.3. Function naming

All functions shall use Achilles and the component name as a prefix, as well as the `fnc_` prefix behind the component name.

Example: `PREFIX_COMPONENT_fnc_functionName`

For Achilles, this is done automatically through the usage of the `PREP` *macro.*

1.1.4. Name case

The only allowed case is camel case.

Correct:

```
private _myVeryLongVariable = "is long";
```

Incorrect:

```
private _MyVerylongVaRiAbLe = "is long";
```

1.2. Files & config

1.2.1. SQF files

Files containing SQF scripts shall have a file name extension of `.sqf`.

1.2.2. Header files

All header files shall have the file name extension of `.hpp`.

1.2.3. Own SQF file

All functions shall be put in their own `.sqf` file.

1.2.4. Config elements

Config files shall be split up into different header files, each with the name of the config and be included in the `config.cpp` of the component.

Example:

```
#include "Achilles_Settings.hpp"
```

Add in `Achilles_Settings.hpp`:

```
class Achilles_Settings {
    // Content
};
```

1.3. String table

All text that shall be displayed to a user shall be defined in a `stringtable.xml` file for multi-language support.

- There shall be no empty string table language values.
- All string tables shall follow the format as specified by Tabler and the translation guidelines form.

1.15.2 2. Macro usage

2.1. Module/PBO specific macro usage

The family of `GVAR` macros define global variable strings or constants for use within a module. Please use these to make sure we follow naming conventions across all modules and also prevent duplicate/overwriting between variables in different modules. The macro family expands as follows, for the example of the module 'balls'.

Macros	Expands to
`GVAR(face)`	`achilles_balls_face`
`QGVAR(face)`	`"achilles_balls_face"`
`QQGVAR(face)`	`""achilles_balls_face""` used inside `QUOTE` macros where double quotation is required.
`EGVAR(leg,face)`	`achilles_leg_face`
`QEGVAR(leg, face)`	`"achilles_leg_face"`
`QQEGVAR(leg, face)`	`""achilles_leg_face""` used inside `QUOTE` macros where double quotation is required.

There also exists the `FUNC` family of macros.

Macros	Expands to
`FUNC(face)`	`achilles_balls_fnc_face` or the call trace wrapper for that function.
`EFUNC(leg, face)`	`achilles_leg_fnc_face` or the call trace wrapper for that function.
`DFUNC(leg, face)`	`achilles_balls_fnc_face` and will **always** be the function global variable.
`LINKFUNC(face)`	`FUNC(face)` or *"pass by reference"* `{_this call FUNC(face)}`
`QFUNC(face)`	`"achilles_balls_fnc_face"`
`QEFUNC(leg, face)`	`"achilles_leg_fnc_face"`
`QQFUNC(face)`	`""achilles_balls_fnc_face""` used inside `QUOTE` macros where double quotation is required.
`QQEFUNC(leg, face)`	`""achilles_leg_fnc_face""` used inside `QUOTE` macros where double quotation is required.

The `FUNC` and `EFUNC` macros shall **not** be used inside `QUOTE` macros if the intention is to get the function name or assumed to be the function variable due to call tracing (see below). If you need to 100% always be sure that you are getting the function name or variable use the `DFUNC` or `DEFUNC` macros. For example `QUOTE(FUNC(face)) == "achilles_balls_fnc_face"` would be an illegal use of `FUNC` inside `QUOTE`.

Using `FUNC` or `EFUNC` inside a `QUOTE` macro is fine if the intention is for it to be executed as a function.

`LINKFUNC` macro allows to recompile function used in event handler code when function cache is disabled, e.g. `player addEventHandler ["Fired", LINKFUNC(firedEH)];` will run updated code after each recompile.

2.1.1. `FUNC` macros, call tracing and non-Achilles/anonymous functions

Achilles implements a basic call tracing system that can dump the call stack on errors or wherever you want. To do this the `FUNC` macros in debug mode will expand out to include metadata about the call including line numbers and files. This functionality is automatic with the use of calls via `FUNC` and `EFUNC`, but any calls to other functions need to use the following macros.

Macros	Example
`CALLSTACK(functionName)`	`[] call CALLSTACK(cba_fnc_someFunction)`
`CALLSTACK_NAMED(function, functionName)`	`[] call CALLSTACK_NAMED(_anonymousFunction,'My anonymous function!')`

These macros will call these functions with the appropriate wrappers and enable call logging into them (but to no further calls inside obviously).

2.2. General purpose macros

CBA script_macros_common.hpp

`QUOTE` is utilized within configuration files for bypassing the quote issues in configuration macros. So, all code segments inside a given config should utilize wrapping in the `QUOTE` macro instead of direct strings. This allows us to use our macros inside the string segments, such as `QUOTE(_this call FUNC(balls))`.

2.2.1. `setVariable`, `getVariable` family macros

Note: These macros are allowed but are not enforced.

Macros	Expands to
`GETVAR(player,MyVarName, false)`	`player getVariable ["MyVarName", false]`
`GETMVAR(MyVarName,objNull)`	`missionNamespace getVariable ["MyVarName", objNull]`
`GETUVAR(MyVarName, displayNull)`	`uiNamespace getVariable ["MyVarName", displayNull]`
`SETVAR(player,MyVarName, 127)`	`player setVariable ["MyVarName", 127]`
`SETPVAR(player,MyVarName, 127)`	`player setVariable ["MyVarNae", 127, true]`
`SETMVAR(MyVarName,player)`	`missionNamespace setVariable ["MyVarName", player]`
`SETUVAR(MyVarName,_control)`	`uiNamespace setVariable ["MyVarName", _control]`

2.2.2 `STRING` family macros

Note that you need the strings in module `stringtable.xml` in the correct format:

```
STR_Achilles_<module>_<string>
```

Example: `STR_Achilles_Balls_Banana`

Script strings (still requires `localize` to localize the string).

Macros	Expands to
`LSTRING(banana)`	`"STR_Achilles_balls_banana"`
`ELSTRING(leg,banana)`	`"STR_Achilles_leg_banana"`

Config strings (requires `$` as the first character):

Macros	Expands to
`CSTRING(banana)`	`"$STR_Achilles_balls_banana"`
`ECSTRING(leg,banana)`	`"$STR_Achilles_leg_banana"`

2.2.3. `PATH` family macros

The family of path macros define global paths to files for use within a module. Please use these to reference files in Achilles. The macro family expands as follows, for the example of the module 'balls'.

Macros	Expands to
`PATHOF(data\banana.p3d)`	`\z\achilles\addons\balls\data\banana.p3d`
`QPATHOF(data\banana.p3d)`	`"\z\achilles\addons\balls\data\banana.p3d"`
`PATHOEF(leg,data\banana.p3d)`	`\z\achilles\addons\leg\data\banana.p3d`
`QPATHOEF(leg,data\banana.p3d)`	`"\z\achilles\addons\leg\data\banana.p3d"`

1.15.3 3. Functions

Functions shall be created in the `functions/` subdirectory, named `fnc_functionName.sqf`. They shall then be indexed via the `PREP(functionName)` macro in the `XEH_preInit.sqf` file.

The `PREP` macro allows for CBA function caching, which drastically speeds up load times.

Note: Beware though that function caching is enabled by default and as such to disable it you need to `#define DISABLE_COMPILE_CACHE` above your `#include "script_components.hpp"` include.

3.1. Headers

Every function should have a header of the following format as the start of their function file:

```
/*
* Author: [Name of Author(s)]
* [Description]
*
* Arguments:
* 0: The first argument <STRING>
* 1: The second argument <OBJECT>
* 2: Multiple input types <STRING|ARRAY|CODE>
* 3: Optional input <BOOL> (default: true)
* 4: Optional input with multiple types <CODE|STRING> (default: {true})
* 5: Not mandatory input <STRING> (default: nil)
*
* Return Value:
* The return value <BOOL>
*
* Example:
* ["something", player] call achilles_common_fnc_myFunction
*
* Public: [Yes/No]
*/
```

Note: This is not the case for inline functions or functions not containing their own file.

3.2. Includes

Every function includes the `script_component.hpp` file just below the function header. Any additional includes or defines must be below this include.

All scripts written must be below this include and any potential additional includes or defines.

3.2.1. Reasoning

This ensures every function starts off in a uniform way and enforces function documentation.

1.15.4 4. Global variables

All global variables are defined in the `XEH_preInit.sqf` file of the component they will be used in with an initial default value.

Note:

Exceptions:

- Dynamically generated global variables.
- Variables that do not origin from Achilles, such as BI global variables or a third party such as CBA.

1.15.5 5. Code style

To help with some parts of the coding style we recommend you get the plugin EditorConfig for your editor. It will help with correct indentations and deleting trailing spaces.

5.1. Curly bracket placement

Curly brackets (`{ }`) which enclose a code block will be the first bracket placed on the line behind the statement in case of `if`, `switch` statements or `while`, `waitUntil` and `for` loops. The second brace will be placed on the same column as the statement and on a separate line.

- Opening brace on the same line.
- Closing brace in own line, the same level of indentation as the keyword.

Correct:

```
class Something: Or {
    class Other {
        foo = "bar";
    };
};
};
```

Incorrect:

```
class Something : Or
{
    class Other
    {
        foo = "bar";
    };
```

Incorrect:

```
class Something : Or {
    class Other {
        foo = "bar";
        };
    };
```

When using `if`/`else`, it is recommended to put `else` on the same line as the closing brace to save space:

```
if (alive player) then {
    player setDamage 1;
} else {
    hint ":(";
};
```

5.1.2. Reasoning

Putting the opening brace in its own line wastes a lot of space, and keeping the closing brace on the same level as the keyword makes it easier to recognize what exactly the brace closes.

5.2. Indents

Note: Indentations consist of 4 spaces. Tabs are not allowed.

Every new scope should be on a new indent. This will make the code easier to understand and read. Spaces are not allowed to trail on a line, the last character needs to be non-blank.

Correct:

```
call {
    call {
        if (/* condition */) then {
            /* code */
        };
    };
};
```

Incorrect:

```
call {
        call {
        if (/* condition */) then
        {
            /* code */
        };
        };
};
```

5.3. Inline comments

Inline comments should use `//`. Usage of `/* */` is allowed for larger comment blocks.

Example:

```
//// Comment    // < incorrect
// Comment      // < correct
/* Comment */   // < correct
```

5.4. Comments in code

Note: All code shall be documented by comments that describe what is being done.

This can be done through the function header and/or inline comments.

Comments within the code shall be used when they are describing a complex and critical section of code or if the subject code does something a certain way because of a specific reason. Unnecessary comments in the code are not allowed.

Correct:

```
// find the object with the most blood loss
_highestObject = objNull;
_highestLoss = -1;
{
    if ([_x] call EFUNC(medical,getBloodLoss) > _highestLoss) then {
        _highestLoss = [_x] call EFUNC(medical,getBloodLoss);
        _highestObject = _x;
    };
} foreach _units;
```

Correct:

```
// Check if the unit is an engineer
(_object getvariable [QGVAR(engineerSkill), 0] >= 1);
```

Incorrect:

```
// Get the engineer skill and check if it is above 1
(_object getvariable [QGVAR(engineerSkill), 0] >= 1);
```

Incorrect:

```
// Get the variable myValue from the object
_myValue = _object getvariable [QGVAR(myValue), 0];
```

Incorrect:

```
// Loop through all units to increase the myvalue variable
{
    _x setvariable [QGVAR(myValue), (_x getvariable [QGVAR(myValue), 0]) + 1];
} forEach _units;
```

5.5. Parentheses around code

When making use of parentheses (( )), use few as possible, if not doing so, you decrease the readability of the code. Avoid statements such as:

```
if (! ((_value))) then { };
```

However, the following is allowed:

```
_value = (_array select 0) select 1;
```

Any conditions in statements shall always be wrapped around brackets.

```
if (! _value) then { };
if (_value) then { };
```

5.6. Negation operator

When using conditions with the negation operator (`!`), we recommend using a space between the value and the operator.

Example:

```
if (! _myVariable) then { };
```

This does not affect the comparison operator:

```
if (_myVariable != _myOtherVariable) then { };
```

5.7. Magic numbers

There shall be no magic numbers. Any magic number shall be put in a `#define` either at the top of the `.sqf` file (below the header) or in the `script_component.hpp` file in the root directory of the component (recommended) in case it is used in multiple locations.

Magic numbers are any of the following:

- A constant numerical or text value used to identify a file format or protocol.
- Distinctive unique values that are unlikely to be mistaken for other meanings.
- Unique values with unexplained meaning or multiple occurrences which could (preferably) be replaced with named constants.

1.15.6 6. Code standards

6.1. Error testing

If a function returns error information, then that error information will be tested.

6.2. Unreachable code

There shall be no unreachable code.

6.3. Function parameters

Parameters of functions must be retrieved through the user of `param` or `params` commands. If the function is part of the public API, parameters must be checked on allowed data types and values through the usage of above-mentioned commands.

Usage of the CBA macro `PARAM_x` or `BIS_fnc_param` is deprecated and not allowed within Achilles.

6.4. Return values

Functions and code blocks that have a specific return value must be a meaningful return value. If it has no meaningful return value, then the function should return `nil`.

6.5. Private variables

All private variables shall make use of the `private` keyword on initialization. When declaring a private variable before initialization, usage of the `private ARRAY` syntax is allowed.

Exceptions to this rule are variables obtained from an array, which shall be done with the usage of the `params` command family, which ensures the variable is declared as private.

Correct:

```
private _myVariable = "hello world";
```

Correct:

```
_myArray params ["_elementOne", "_elementTwo"];
```

Incorrect:

```
_elementOne = _myArray select 0;
_elementTwo = _myArray select 1;
```

6.6. Lines of code

Any function shall contain no more that 250 lines of code, excluding the function header and any includes.

6.7. Variable declarations

Declarations should be at the smallest feasible scope.

Correct:

```
if (call FUNC(myCondition)) then {
    private _areAllAboveTen = true; // <- smallest feasable scope

    {
        if (_x >= 10) then {
            _areAllAboveTen = false;
        };
    } forEach _anArray;

    if (_areAllAboveTen) then {
        hint "all values are above ten!";
    };
};
```

Incorrect:

```
private _areAllAboveTen = true; // <- this is bad, because it can be initialized in␣
↪the if statement
if (call FUNC(myCondition)) then {
    {
        if (_x >= 10) then {
            _areAllAboveTen = false;
        };
    } forEach _anArray;

    if (_areAllAboveTen) then {
        hint "all values are above ten!";
    };
};
```

6.8. Variable initialization

Private variables will not be introduced until they can be initialized with meaningful values.

Correct:

```
private _myVariable = [1, 2] select _condition;
```

Correct:

```
private _myVariable = 0; // good because the value will be used
{
    _x params ["_value", "_amount"];
    if (_value > 0) then {
        _myVariable = _myVariable + _amount;
    };
} forEach _array;
```

Incorrect:

```
private _myvariable = 0; // Bad because it is initialized with a zero, but this value␣
↪does not mean anything
if (_condition) then {
    _myVariable = 1;
} else {
    _myvariable = 2;
};
```

6.9. Initialization expression in `for` loops

The initialize expression in a `for` loop shall perform no actions other than to initalize the value of a single `for` loop parameter.

6.10. Increment expression in `for` loops

The increment expression in a `for` loop shall perform no action other than to change a single loop parameter to the next value for the loop.

6.11. Usage of `getVariable`

When using `getVariable`, there shall either be a default value given in the statement or the return value shall be checked for correct data type as well as the return value. A default value may not be given after a `nil` check.

Correct:

```
_return = object getVariable ["varName", 0];
```

Correct:

```
_return = object getVariable "varName";
if (isNil "_return") exitWith {};
```

Incorrect:

```
_return = _obj getVariable "varName";
if (isNil "_return") then { _return = 0; };
```

6.12. Global variables

Global variables should not be used to pass along information from one function to another. Use arguments instead.

Correct:

```
fnc_example = {
    params ["_content"];
    hint _content;
};

["hello my variable"] call fnc_example;
```

Incorrect:

```
fnc_example = {
    hint GVAR(myVariable);
};

GVAR(myVariable) = "hello my variable";
call fnc_example;
```

6.13. Temporary objects and variables

Unnecessary temporary objects or variables should be avoided.

6.14. Commented out code

Code that is not used (commented out) shall be removed.

6.15. Constant global variables

There shall be no constant global variables, constants shall be put in a `#define`.

6.16. Logging

Functions shall whenever possible and logical, make use of logging functionality through the logging and debugging macros from CBA and Achilles.

6.17. Constant private variables

Constant private variables that are used more than once shall be put in a `#define`.

6.18. Code used more than once

Any piece of code that could/is used more than once, shall be put in a function and it's separate `.sqf` file unless this code is less as 5 lines and used only in a *per-frame handler*.

1.15.7 7. Design considerations

7.1. Readability vs performance

This is an open source project that will have different maintainers over its lifespan. When writing code, keep in mind that other developers will also need to understand your code. Balancing readability and performance is a non-black and white subject.

The rule of thumb is:

- When improving the performance of code that sacrifices readability (or vice-versa), first see if the design of the implementation is done in the best way possible.
- Document that change with the reasoning in the code.

7.2. Scheduled vs unscheduled

Note: Avoid the usage of scheduled space as much as possible and stay in unscheduled.

This is to provide a smooth experience to the user by guaranteeing code to run when we want it. See *Performance considerations, spawn and execVM* for more information.

This also helps avoid various bugs as a result of unguaranteed execution sequences when running multiple scripts.

7.3. Event-driven

All Achilles components shall be implemented in an event-driven fashion. This is done to ensure code only runs when it is required and allows for modularity through low coupling components.

Event handlers in Achilles are implemented through the CBA event system. They should be used to trigger or allow triggering of specific functionality.

More information on the CBA events system and CBA player events pages.

Warning: BI's event handlers (`addEventHandler`, `addMissionEventHandler`) are slow when passing a large code variable. Use a short code block that calls the function you want.

```
player addEventHandler ["Fired", FUNC(handleFired)]; // bad
player addEventHandler ["Fired", {call FUNC(handleFired)}]; // good
```

7.4. Hashes

When a key-value pair is required, make use of the hash implementation from Achilles.

Hashes are a variable type that store key value pairs. They are not implemented natively in SQF, so there are a number of macros and functions for their usage in Achilles. If you are unfamiliar with the idea, they are similar in function to `getVariable` / `setVariable` but do not require an object to use.

The following example is a simple usage using our macros which will be explained further below.

```
_hash = HASHCREATE;
HASH_SET(_hash,"key","value");

if (HASH_HASKEY(_hash,"key")) then {
    player sideChat format ["val: %1", HASH_GET(_hash,"key"); // will print out "val:␣
↪value"
};

HASH_REM(_hash,"key");

if (HASH_HASKEY(_hash,"key")) then {
    // this will never execute because we removed the hash key/value pair "key"
};
```

A description of the above macros is below.

Macros	Usage
`HASHCREATE`	Used to create an empty hash.
`HASH_SET(hash,key,val)`	Will set the hash key to that value, a key can be anything, even objects.
`HASH_GET(hash,key)`	Will return the value of that key (or `nil` if it doesn't exist).
`HASH_HASKEY(hash,key)`	Will return `true`/`false` if that key exists in the hash.
`HASH_REM(hash,key)`	Will remove that hash key.

7.4.1. Hash lists

A hash list is an extension of a hash. It is a list of hashes!

The reason for having this special type of storage container rather than using a normal array is that an array of normal hashes that are similar will duplicate a large amount of data in their storage of keys. A hash list, on the other hand, uses a common list of keys and an array of unique value containers.

The following will demonstrate their usage.

```
_defaultKeys = ["key1", "key2", "key3"];
// create a new hashlist using the above keys as default
_hashList = HASHLIST_CREATELIST(_defaultKeys);

//lets get a blank hash template out of this hashlist
_hash = HASHLIST_CREATEHASH(_hashList);
```

(continues on next page)

(continued from previous page)

```
//_hash is now a standard hash...
HASH_SET(_hash,"key1","1");

//to store it to the list we need to push it to the list
HASHLIST_PUSH(_hashList, _hash);

//now lets get it out and store it in something else for fun
//it was pushed to an empty list, so it's index is 0
_anotherHash = HASHLIST_SELECT(_hashList,0);

// this should print "val: 1"
player sideChat format["val: %1", HASH_GET(_anotherHash,"key1")];

//Say we need to add a new key to the hashlist
//that we didn't initialize it with? We can simply
//set a new key using the standard HASH_SET macro
HASH_SET(_anotherHash,"anotherKey","another value");
```

As you can see above, working with hash lists is fairly simple, a more in-depth explanation of the macros is below.

Macros	Usage
`HASHLIST_CREATELIST(keys)`	Creates a new hash list with the default keys, pass `[]` for no default keys.
`HASHLIST_CREATEHASH(hashlist)`	Returns a blank hash template from a hash list.
`HASHLIST_PUSH(hashlist,hash)`	Pushes a new hash into the end of the list.
`HASHLIST_SELECT(hashlist, index)`	Returns the hash at that index in the list.
`HASHLIST_SET(hashlist,index, hash)`	Sets a specific index to that hash.

Note: Hashes and hash lists are implemented with SQF arrays, and as such, they are passed by reference to other functions. Remember to make copies (using the + operator) if you intend for the hash or hash list to be modified without the need for changing the original value.

1.15.8 8. Performance considerations

8.1. Adding elements to arrays

When adding new elements to an array, `pushBack` shall be used instead of the binary addition or `set`. When adding multiple elements to an array `append` may be used instead.

Correct:

```
_array pushBack _value;
```

Correct:

```
_array append [1, 2, 3];
```

Incorrect:

```
_array set [count _array, _value];
_array = _array + [_value];
```

When adding an new element to a dynamic location in an array or when the index is pre-calculated, `set` may be used.

When adding multiple elements to an array, the binary addition may be used for the entire addition.

8.2. `createVehicle`

`createVehicle` array shall be used.

8.3. `createVehicle(local)` position

`createVehicle(local)` used with a non-`[0, 0, 0]` position shall be used, except on `#` objects (e.g. `#lightsource`, `#soundsource`) where empty position search is not performed.

This code requires ~1.00 ms and will be higher with more objects near the wanted position:

```
_vehicle = _type createVehicleLocal _posATL;
_vehicle setPosATL _posATL;
```

While this one requires ~0.04 ms:

```
_vehicle = _type createVehicleLocal [0, 0, 0];
_vehicle setPosATL _posATL;
```

8.4. Unscheduled vs scheduled

All code that has a visible effect for the user or requires time specific guaranteed execution shall be written in unscheduled space.

8.5. Avoid `spawn` and `execVM`

`execVM` and `spawn` are to be avoided wherever possible.

8.6. Empty arrays

When checking if an array is empty `isEqualTo` shall be used.

8.7. `for` loops

```
for "_y" from # to # step # do { ... }
```

shall be used instead of

```
for [{ ... }, { ... }, { ... }] do { ... };
```

whenever possible.

8.8. `while` loops

While is only allowed when used to perform an unknown finite amount of steps with unknown or variable increments. Infinite `while` loops are not allowed.

Correct:

```
_original = _object getvariable [QGVAR(value), 0];

while {_original < _weaponThreshold} do {
    _original = [_original, _weaponClass] call FUNC(getNewValue);
}
```

Incorrect:

```
while {true} do {
    // anything
};
```

8.9. `waitUntil`

The `waitUntil` command shall not be used. Instead, make use of CBA's `CBA_fnc_waitUntilAndExecute`.

```
[{
params ["_unit"];
_unit getVariable [QGVAR(myVariable), false]
},
{
    params ["_unit"];
    // Execute any code
}, [_unit]] call CBA_fnc_waitUntilAndExecute;
```

1.16 Creating Documentation

Contents

To get started with contributing to the Achilles Lexicon, we have to do some first steps in case you don't have the tools or if you do, then you can jump right in, but before so, get acquainted with the guidelines we follow.

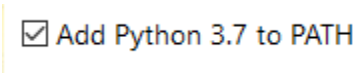
1.16.1 1. Installation

Note: If you have any of these already installed then you can skip that step.

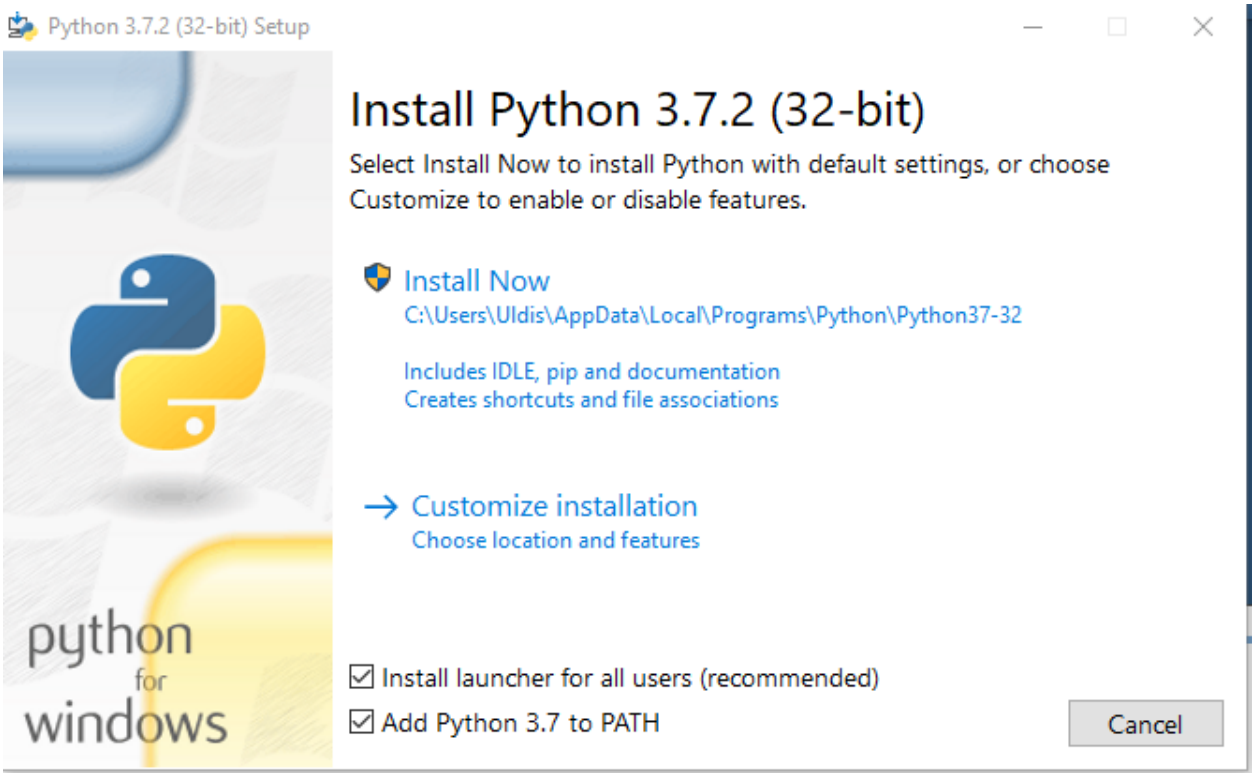
1.1. Python 3

To get started, we need to install Python 3. Dowload the latest version of Python at the Python website.

When installing Python, remember to check *Add Python 3.x to PATH*



Choose the default install option - *Install Now*



1.2. Git Bash

To get the documentation and later save your changes and upload them to GitHub, we need Git.

To download Git, go to the Git project website.

Follow the default installer options and you should be fine.

1.3. Sphinx

To install Sphinx open Git Bash and run the following command:

```
pip install sphinx
```

Tip: If running the above command, you experience and error: try running Git Bash as administrator.

1.4. Make

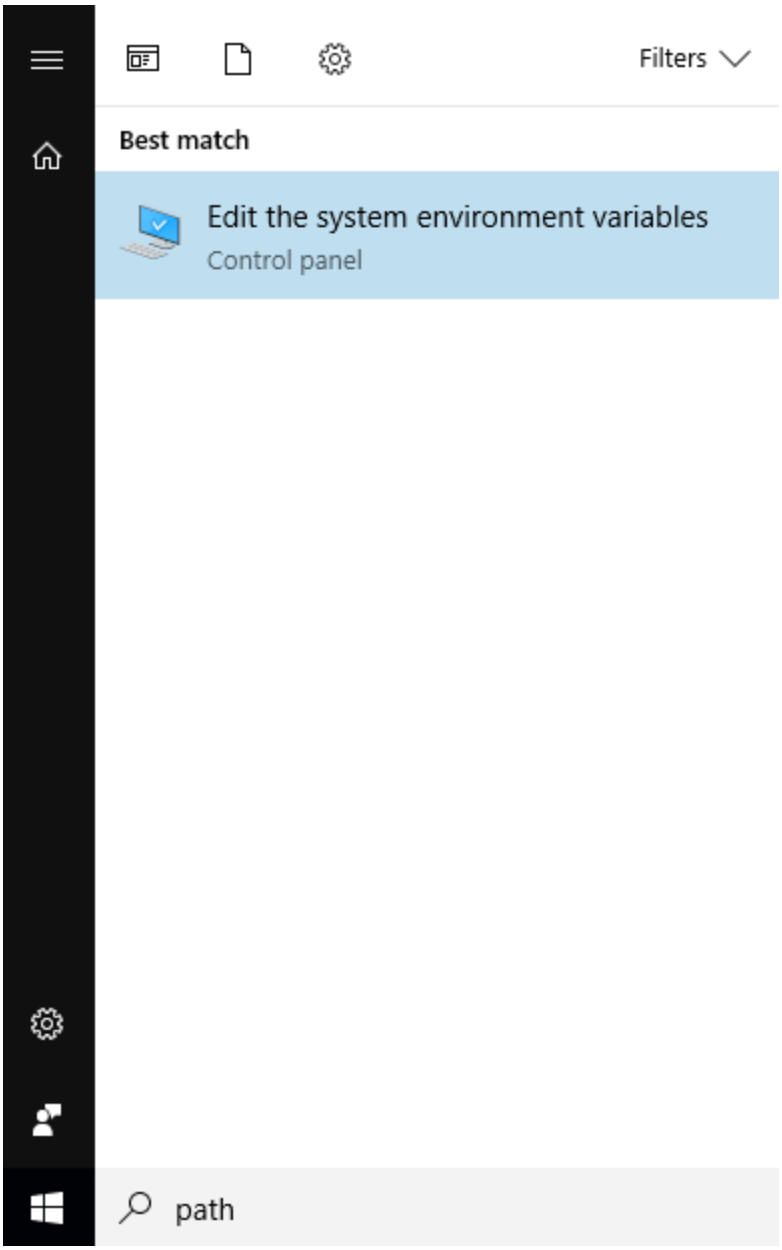
We use Make for easy building of documentation which can also be used to display any errors.

Download Make from here. Install it following all the defaults.

Once installed, you should add the Make directory path to the `PATH` system variable.

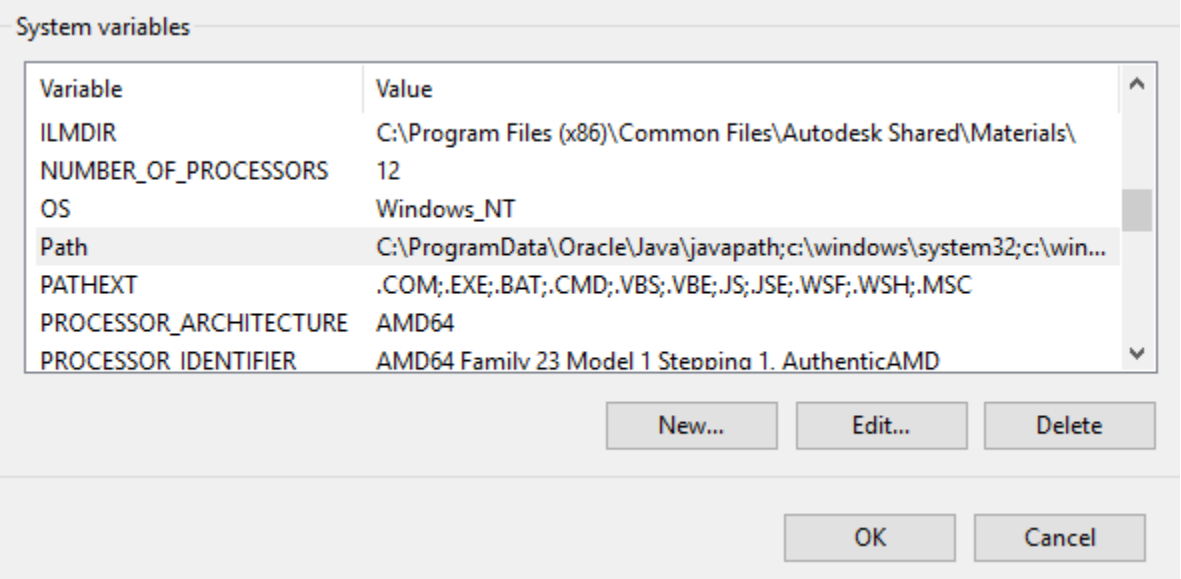
1.4.1. Adding Make to the `PATH` variable

1. Find the Edit the system environment variables in Control Panel or in Search for Windwos 10 machines.



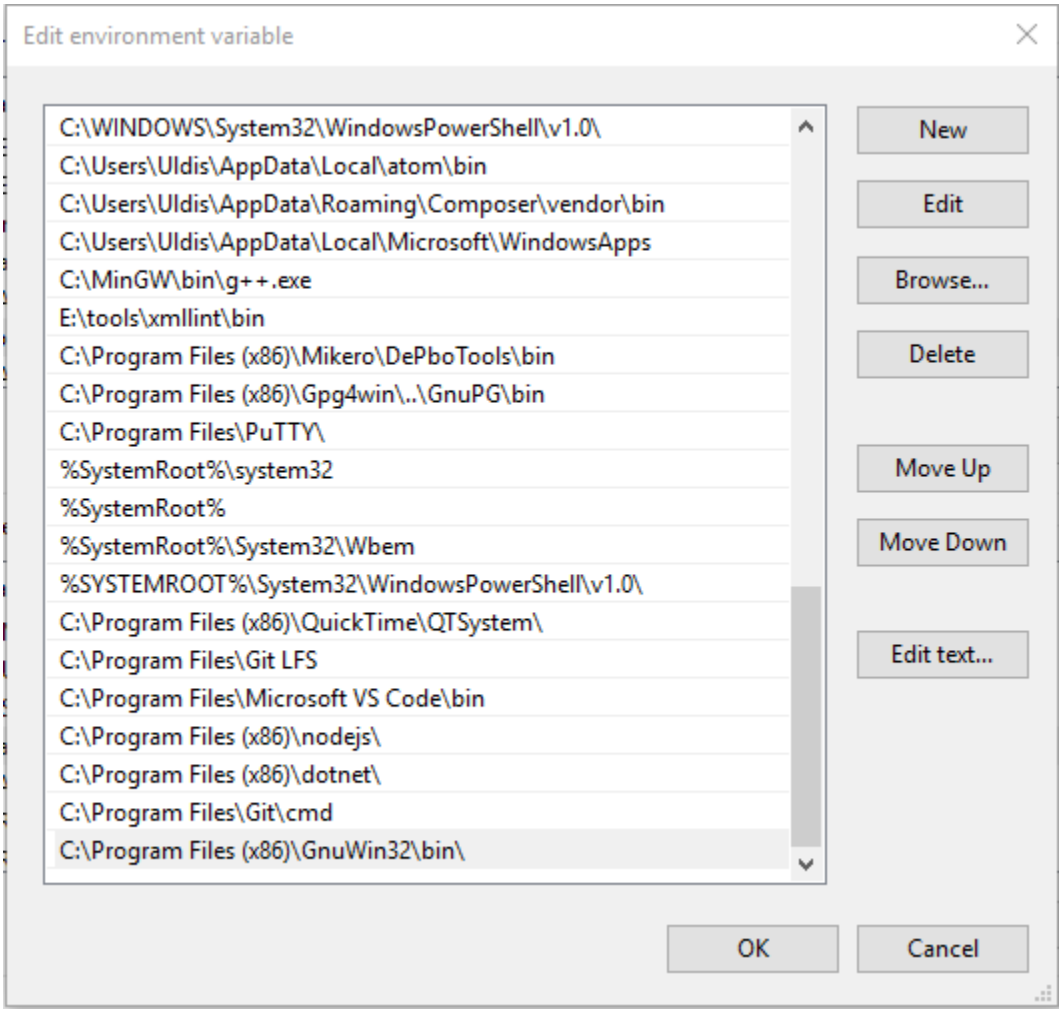
2. Click on the environment variables button.

3. Find the Path variable under the System variables and press Edit.



4. Press New and enter the installation directory of Make and add `bin` to the path and save.

```
C:\Program Files (x86)\GnuWin32\bin\
```



5. After this you should be able to run `make` in Git Bash and you should see the following:

```
make
make: *** No targets specified and no makefile found.  Stop.
```

Note: If you run the `make` command in Git Bash and see the following:

```
make
bash: make: command not found
```

Restart all open Git Bash'es to refresh the loaded `PATH` variable.

1.16.2 2. Editor set up

2.1. Visual Studio Code

If you are using Visual Studio Code, we recommend using the following extensions to provide a better documentation experience.

- reStructuredText for editing restructured text and having a preview in-editor.

1.16.3 3. Cloning the repository

Tip: To quickly open up the directory you want in Git Bash, you can use the context menu (Right Mouse Button) in Windows Explorer and click `Git Bash Here`.

To get Achilles, you need to **fork** Achilles and open up Git Bash and run the following commands:

```
git clone https://github.com/YourUserNameHere/Achilles/
cd Achilles/
git checkout rewrite
git checkout -b aMeaningfulBranchNameHere
```

1.16.4 4. Building documentation

To build documentation locally and to test for errors, run the following in Git Bash:

```
cd docs/
make html
```

If documentation was built without warnings and errors, then you can create a commit and push to GitHub:

```
git add .
git commit -m "Enter a meaningful commit message here"
git push origin aMeaningfulBranchNameHere
```

Afterwards, go into GitHub and create a pull request.

1.16.5 5. Documentation guidelines

5.1. Table of contents

Each page should have a table of contents, which is a restructured text directive called `.. contents::`.

5.2. Numbering system

All pages must contain numbered headings with the format: `x.x.x.`.

Note: Note the last dot after all the numbers.

The deepest level you can go into is 3 levels deep (`x.x.x.`).

5.3. Headings

Headings are formatted as follows:

```
Page title (section) (will be displayed in the left sidebar)
============================================================

Typically, here you would insert the table of contents.

1. Subsection
-------------

This is a subsection.

1.1. Subsubsection
^^^^^^^^^^^^^^^^^^

This is a subsubsection.

1.1.1. Paragraph
""""""""""""""""

This is a paragraph.

1.1.2. Another paragraph
""""""""""""""""""""""""

This is another paragraph.

2. Different subsection
-----------------------

This is a completely different subsection.
```

Warning: Notation under headings are to be the exact length as the title, otherwise it is considered a error.

5.5. Naming

All pages, sections, subsections, etc. are to have a meaningful name and should closely represent the actual content under the heading.

Page titles should not start with numbers unless absolutely necessary.

Headings should avoid using complicated names that would be difficult to understand by inexperienced users.

Headings are to be short and concise without being overly long to prevent issues on mobile devices.

Headings are to start with a capital letter but then follow **lowercase**.

Note: In case there is an acronym (should be avoided), a name or anything else that should start with a capital letter, then it is to be considered an exception to the naming rule above.

5.5. Images

To declutter the directory structure, images should be in separate directories, where only images are stored for only one page.

1.16.6 6. Restructured text cheat sheet

A useful and more comprehensive cheat sheet for restructured text can be found on Thomas Cokelaer's website.

1.17 Installation

1.18 Translating Achilles

1.19 Contribute

1.20 Roadmap

1.21 Team