OFP Preprocessor Explained

I. Introduction

1. Who is this document for?

For those writing custom configurations and dialog files for **Operation Flashpoint**. Here you'll find explanation of the preprocessor directives which will improve your code.

2. What is a preprocessor?

It is a program that prepares (transforms) code for further interpretation / compilation. Modification occurs in the memory and the file is not overwritten.

OFP preprocessor is a stripped down version of **C preprocessor** (similar syntax, less capabilites).

3. Which files are being preprocessed?

Every file with **C**-like syntax: *config.cpp*, *resource.cpp*, *description.ext*, *mission.sqm* and so on. Also, files loaded using `preProcessFile` command.

4. When the files are preprocessed?

- Global configuration and resource, addon configs, user settings – during the launch of the game.
- *mission.sqm*, *description.ext* – every time you open the mission. In **Mission Editor** – every time you load / save the mission.
- Files loaded with `preProcessFile` – when that command is executed.

II. Commands

1. Syntax

- Directive must be written in a new line, preceded only by a whitespace.
- Directive starts with a '#' (hash) character.
- All directives must be written in lower case.

2. Order of execution

Preprocessor analyzes the code line by line and does not retract. For example, a macro written before it was defined won't work (more on macros later).

3. Comments

Comments (text used to describe a code) are indicated by:

```
// double slash - single comment line
/* slash asterisk – comment block */
```

They are removed by the preprocessor so the game does not "see" them.

```
// human class
class MySoldier
{
	/* his properties:
	- super armor */
	armor = 500
};
```

4. #include

Copies the code from a target file and pastes it where `#include` directive is.

```
#include "codestrip.hpp"
#include <codestrip.txt>
```

Brackets are equal to quotation marks.

Source directory is:

- for *description.ext*, addon configs – game root folder (where *exe* file is)
- for global config and resource – their source folder

Alternatively you may write a path starting from drive:

```
#include "d:\temp\codestrip.txt"
```

To move to parent directory use '`..`' (two dots):

```
#include "..\codestrip.txt"
```

Addons location is saved to memory. To include file from one of them write:

```
#include "<addon folder name>\<file>"
```

OFP preprocessor **does not** support computed includes (macro for file name).

```
#define path "codestrip.txt"
#include path
```

This code will cause an error. Macros will be explained later.

How to make *description.ext* include one of the mission files:

- Mission is not packed and is located in the user folder.

```
#include "Users\<user name>\missions\<mission>\<file>"     // SP
#include "Users\<user name>\mpmissions\<mission>\<file>"   // MP
```

- Mission is packed (PBO) and is located in the SP / MP missions folder.

```
#include "missions\__cur_sp.<island>\<file name>"          // SP
#include "mpmissions\__cur_mp.<island>\<file name>"        // MP
```

5. #define

Creates a macro (alias, shortcut if you prefer). Macros written in the code replace (expand) themselves with the text assigned to them. Syntax:

```
#define <name> <content>
```

For example:

```
#define dialogX 0.5
x = dialogX
```

After preprocessing:

```
x = 0.5
```

Names consist of alphanumeric characters, starting with a letter.

```
#define 10     // wrong
#define a10    // correct
```

Other characters count as content.

```
#define a10.a
```

In this example '`a10`' is the name and content is '`.a`'.

Names are case sensitive.

```
#define something 1
#define Something 2
```

It's possible to give the macro a class property name.

```
#define armor 1
armor = 1
```

It will be replaced with:

```
1 = 1
```

Define your macros carefully.

You may assign multiple lines with '\' (backslash).

```
#define properties x=0.5; \
y=0.5; \
w=0.2; \
h=0.2;
```

This expands into a single line:

```
x=0.5; y=0.5; w=0.2; h=0.2;
```

Backslash must be the last character in the line. Otherwise it won't work.

Macro name inside a quote is ignored by the preprocessor.

```
#define something 10
text = "something"
```

Game will display *something*, not *10*.

Defining the same macro again updates it.

```
#define something 1
#define something 10
x = something
```

It will expand into:

```
x = 10
```

You can place a macro inside other macro.

```
#define something    cake
#define shopping     buy something
shopping
```

Results in:

```
buy cake
```

Preprocessor looks for other macros at the moment of expanding.

6. #define with arguments

There are two types of macros:

- Object-like (replaces A with B)
- Function-like (replaces A with B with given arguments)

Function-like macro is defined by adding parenthesis:

```
#define myFunction()
#define add(argument1, argument2) argument1 + argument2
add(2,2)
```

Output:

```
2 + 2
```

It's not possible to have both object-like and function-like macros with the same name. The new one simply replaces the old one.

Arguments are local macros themselves. They cannot be replaced by a global macro.

```
#define arg1 0
#define add(arg1,arg2)  arg1 + arg2
add(2,2)
```

The result:

```
2 + 2
```

Number of the passed arguments must be accurate. Otherwise the macro won't work. You may leave any argument empty but you'll have to add commas in between.

```
#define myFunction(arg1,arg2,arg3)
myFunction(,2,)
```

Other macros can be passed as arguments, including function-like ones:

```
#define add(arg1,arg2)  arg1 + arg2
add( add(2,2),2 )
```

Expands into:

```
2 + 2 + 2
```

Every passed character is included – arguments are not formatted by the preprocessor.

```
#define Display(arg1,arg2,arg3) arg1,arg2,arg3

Display(   space   ,/* comment */,
new line)
```

Output:

```
   space   ,/* comment */,
new line
```

7. # - stringification operator

To be used in macro definitions. '#' (single hash) operator wraps the text with quotation marks. Syntax:

```
#<text>
```

Example:

```
#define stringify(argument)  #argument
text = stringify(semi;colon)
```

Expands into:

```
text = "semi;colon"
```

8. ## - concacentation operator

To be used in macro definitions. You can merge two macros or text and a macro together using '##' (double hash). It works like '+' operator in **C**-like languages. Syntax:

```
<text1>##<text2>
```

Example of merging macro arguments and underscore:

```
#define glue(arg1,arg2)  arg1##_##arg2
glue(Adam,Smith)
```

Output:

```
Adam_Smith
```

9. #undef

Removes a macro. Syntax:

```
#undef <name>
```

Example:

```
#define something 4
#undef something
x = something
```

Result:

```
x = something
```

This directive has no effect if given argument is not a macro.

Non-alphanumeric characters are ignored but not removed from the file. They will most likely cause a config syntax error.

```
#undef something...
```

Don't write parenthesis when erasing function-like macro.

```
#undef myFunction(arg1)       // wrong
#undef myFunction             // correct
```

10. #ifdef, #endif – conditional inclusion

OFP preprocessor supports conditional directives but only those related to macros. `#ifdef` checks if given macro has been defined. If so, the preprocessor includes all instructions until it encounters `#endif`. Syntax:

```
#ifdef <name>
<code block>
#endif
```

Block may contain standard code and other directives as well. Example:

```
#define something 10

#ifdef something
	#define adam smith
	class mySoldier
	{
		armor = 50
		displayName = adam
	};
#endif
```

Result:

```
class mySoldier
{
	armor = 50
	displayName = smith
};
```

It's possible to nest conditions.

```
#ifdef rocket_launcher

	#ifdef magazine_name
		magazines[] = { magazine_name, magazine_name };
	#endif

#endif
```

`#ifdef` treats non-alphanumeric characters like the macros do.

```
#ifdef a10.a
```

Here `a10` is the condition and `.a` is a part of the code block.

11. #ifndef

It's similar to `#ifdef` but checks if the macro **has not** been defined.

```
#ifndef something
        x = 0
#endif
```

12. #else

To be used in condition code block. Instructions written after this directive are included if the condition has not been met. Example:

```
#ifdef something
        x = 1
#else
        x = 0
#endif
```

Result:

```
x = 0
```

III. Usage

1. Comments

Adding code description is very important. You may perfectly comprehend structure of your work now but after some time you'll forget it. Also other people viewing your files may have a better chance to learn something if you'll leave some information.

Write comments to describe elements / mechanics of your code.

2. Including files

It's very convenient to break big files into smaller pieces, each holding classes of one type.

```
Config.cpp:

        #include "CfgPatches.hpp"
        #include "CfgModels.hpp"
        #include "CfgVehicles.hpp"
```

Other application could be to have files that read global user settings.

```
Operation Flashpoint\UserSettings.hpp:

        #define human_armor 500

Addon Config.cpp:

        #include "UserSettings.hpp"
        armor = human_armor
```

3. Macros

Macros are used to replace repeatable segments of the code. Object-like macros create constant values. For example:

```
#define human_armor 500
```

Now if you'll use it in every soldier class...

```
class mySoldier
{
	armor = human_armor
};
```

... it will be very easy to adjust it, without the need to poke every value.

Function-like macros are used to shorten similar but not identical chunks of code. For example defining soldier classes:

```
#define MakeSoldier( CLASSNAME, DISPLAYNAME ) \
class CLASSNAME : soldierWB \
{ \
	displayName = DISPLAYNAME; \
	armor = human_armor; \
};

MakeSoldier( my_regular, "Regular" )
MakeSoldier( my_sniper, "Sniper" )
MakeSoldier( my_engineer, "Engineer" )
```

4. Conditions

Conditions let you send alternative version of your code to the game. For example: you can set up user custom options or a debug mode.

```
#define Difficulty_Easy

	#ifdef Difficulty_Easy
		#define human_armor 500
	#endif


	#ifdef Difficulty_Hard
		#define human_armor 100
	#endif
```

You may also keep old parts of the code in case you'll need to revert.

```
#define old_version

#ifdef old_version
	class oldSoldier {};
#else
	class newSoldier {};
#endif
```

IV. Errors

If you'll make a mistake the game process will be terminated and message with error code will appear.

0 – Incorrect condition

Missing `#endif` after condition statement.

```
#ifdef something
       x = 1
```

1 – Incorrect include

Preprocessor could not add file you selected with `#include` directive because it doesn't exist OR it's not a text file.

2 – Incorrect include

Missing brackets after `#include` directive.

```
#include something
#include
```

4 – Incorrect macro

Macro argument definition is wrong because:

- there is a non-alphanumeric character inside parenthesis.
- name does not start with a letter.
- parenthesis has not been closed.

```
#define myFunction(a..) 2 + a..      // illegal character

#define myFunction(10) 2 + 10        // starts with number

#define myFunction(arg1,             // missing bracket
```

6 – Incorrect condition

Missing condition statement before `#endif`.

```
        armor = 1
#endif
```

7 – Unknown directive

You have entered a command which is not supported by the preprocessor.

```
#something
#DEFINE something 10
```

11 – Incorrect argument

Condition argument OR `#undef` argument is wrong because:

- name does not start with a letter
- argument is missing

```
#undef 10a      // starts with number
#ifdef          // no argument
#ifdef .a10     // no argument, .a10 counts as content
```

12 – Incorrect condition

Missing `#endif` after `#else`.

```
#ifdef something
        x = 1
#else
        x = 0
```

V. Afterword

Research source:

http://gcc.gnu.org/onlinedocs/cpp/index.html

The preprocessor gives you potential to write shorter and more flexible code.

Have fun

Faguss (ofp-faguss.com)