Config syntax

Basic syntax

As for traditional INI files, all the config information is expressed using key/value pairs.
These pairs are then organized into sections.

A section is declared with square brackets ([ ]). All the key/value pairs defined below will be part of this section.
The section ends when another one begins.

```
[Section]
```

Keys and values are delimited by an equals sign (=).

```
Key = Value
```

Semicolon (;) indicates the start of a comment. Comment continue till the end of the line.

```
; This is a comment and will be ignored by the config module
```

Sections can be defined in more than one place. They can even span on multiple config files.

Here's a simple example.

```
; Here's a basic example of orx's config syntax
[MySection] ; This defines the start of 'MySection'
MyKey      = MyValue; Here we give a value for 'MyKey'
MyOtherKey = MyOtherValue; And here we give one for 'MyOtherKey'

[MyOtherSection]; Here ends 'MySection' and we're now in 'MyOtherSection'
AKey = Yet Another Value

[MySection]; Here we're back to 'MySection'
MyLastKey = MyLastValue; Adding another key/value pair to 'MySection'
```

NB: Spaces around the assign operator ('=') are trimmed and will simply be ignored by the config system.

If you want to use a ';' as part of a non-numerical value, you need to use a block assignation. Blocks are delimited with double quotes '"'. Blocks are also the only way to have values covering multiple lines.

```
MyKey      = "MyValuePart1 ; MyValuePart2"
MyOtherKey = "This value
spans
on multiple lines"
```

If you double the first '"', the string won't be considered as a block but as a normal value and will have a '"' as start of the value.

```
MyKey = ""MyQuotedValue"
```

Here the string "MyQuotedValue" (including the double quotes), will be stored as a value for the key 'MyKey'.

Inheritance

The inheritance system is based on the same idea than inheritance in object-oriented programming.

The basic idea is that all the keys defined in a section can be inherited by any other section [1].
The inheriting section (aka the child section) can then add new keys or even override any key defined in the parent section.

In order to do so, the arobase ('@') is used as an inheritance marker.

```
[Parent]
MyKey1 = MyValue1
MyKey2 = MyValue2

[Child@Parent]; <= The 'Child' section now contains all key/value pairs
defined in the 'Parent' section
```

If you don't want to inherit a whole section, you can also use inheritance directly for a single key.
If the parent key doesn't have the same name as the child one, you can specify its complete name using the dot ('.') separator in addition to the inheritance marker.

```
[Parent]
MyKey      = MyValue
MyOtherKey = MyOtherValue

[Child]
MyKey     = @Parent; <= The value for 'MyKey' will be inherited from the one
defined in the 'Parent' section, with the same key name.
MyLastKey = @Parent.MyKey; <= The value for 'MyLastKey' will also be
inherited from the key 'MyKey' defined in the 'Parent' section.
```

In the previous example, we see that both 'MyKey' and 'MyLastKey' of the 'Child' section inherit from 'MyKey' of the 'Parent' section.
In this example, the key 'MyOtherKey' won't be inherited in the section 'Child'.

When using inheritance, when the parent key's value change, even at runtime, the child value will also be changed.
Inherited values can be chained as seen in the example below.

```
[GrandParent]
```

```
MyKey      = MyValue
MyOtherKey = MyOtherValue

[Parent]
MyKey = @GrandParent

[Child@Parent]
```

In the section 'Child', there's only one key defined called 'MyKey'. Its value is inherited from the 'Parent' section who already inherits it from the 'GrandParent' one. In other words, when `GrandParent.MyKey` changes, both `Parent.MyKey` and `Child.MyKey` will be affected.

All values can implicitly refer to their own section using the inheritance marker '@' by itself. Its value will be dynamic and carry and inheritance to always result in the name of the child section.

A parent can be removed when overriding a section and using no name after '@'. The implicit default parent section can be ignored using the specific '@@' syntax.

```
[Object] <= This section doesn't use any explicit parent but will use the
implicit default parent if defined
[Object@Template] <= This section now uses 'Template' as an explicit parent
section;
[Object@] <= This section now has removed any explicit parent but is still
using the implicit default parent if defined
[Object@@] <= This section now has removed any parent and will not use the
implicit default parent section either
```

To learn more about the default parent section, please refer to the Config section of the main settings page.

```
[Template]
MyKey = @; <= The value for 'MyKey' will be 'Template', the name of this
section
MyOtherKey = @; <= Same here, the value for 'MyOtherKey' will also be the
name of this section: 'Template'

[Object@Template]
MyNewKey = @; <= The value for 'MyNewKey' will be 'Object'
MyKey = @Template; <= The value for 'MyKey' will be inherited from the
section 'Template' using the same key and its value will be 'Object', ie.
this section, not the parent one, 'Template'
```

MyOtherKey will also use the section inheritance and its value will be 'Object', ie. this section, not the parent one, 'Template'.

Includes

A config file can include any number of other files. This helps keeping config files short and focused on one aspect of your game.

For example, you can define all your UI objects/config properties in a file called ui.ini and so on. Here's the syntax.

```
@path/to/MyIncludedFile@
```

NB: The path used is relative to the current working directory and not to the path of the config file that contains it!

Please also note that the include will be done in-place. This means that any values defined in the included file might override any previously defined values. In the same way, any values defined in the included files can then be overridden afterwards.

When writing any key/value pair after the include without redefining a new section, the last section defined before the include will be used.

```
[MySection]
Key1 = Var1

@IncludeFile@

Key2 = Var2; <= this will still be added to the 'MySection' section
```

Numerical values

Basic types

Boolean values are expressed with literals `true` and `false`:

```
VSync = true
```

Floating points (float) values are expressed using a decimal separator ('.').

```
MyFloat = 3.5
```

Integers can be expressed in the most common bases using prefixes:

- Decimal, using no prefix

```
DecimalValue = 16
```

- Hexadecimal, prefix '0x'

```
HexadecimalValue = 0x10
```

- Octal, prefix '0'

```
OctalValue = 020
```

- Binary, prefix '0b'

```
BinaryValue = 0b10000
```

Vector

Vectors are always defined using **three** components, separated by commas (',') and enclosed by either round brackets ('()') or curly brackets ('{}'), with no distinction.

```
MyVector      = (1.0, 2.0, 3.0)
MyOtherVector = {4, 5, 6}
```

Vectors are used, most of the time, to represent coordinates (x, y, z) or color components (r, g, b).

Random

Whenever numerical values are used [2], a random generated [3] value can be obtained using the tilde character ('~') to separate the low and the high boundaries.

```
RandomInt    = 1 ~ 10
RandomFloat  = 0.5 ~ 1.0
RandomVector = (0.0, 0.0, 0.0) ~ (1.0, 1.0, 1.0)
```

Every time the value for a random defined key is queried in the code, a new random value between the specified boundaries will be generated.

Lists

Lists of values for a single key are also supported. All the elements are separated using the pound character ('#'). Lists can also contains random values as elements.

```
ValueList = Val1 # Val2 # RandVal3 ~ RandVal4 # Val5
```

If spaces are defined around the list separators '#', they will be ignored.

NB: Spaces around the list delimiter ('#') are trimmed and will simply be ignored by the config system.
NB: When querying lists using the config API, please look at the functions containing the word 'List' in their name. If you use a non-list function to query a key that contains a list value, any element of the list will be randomly returned.

But let's visualize this with examples. 
Code

```
orxVECTOR vMyVector;
orxConfig_GetVector("MyVector", &vMyVector);
```

```
orxFLOAT fMyFloat = orxConfig_GetFloat("MyFloat");
```

INI

```
MyVector = (1, 2, 3) # (4.0, 5.0, 6.0)
MyFloat = 1.0 # 2.0 # 3.0
```

The functions will then randomly return (1, 2, 3) or (4.0, 5.0, 6.0) for the vector one, and 1.0, 2.0 or 3.0 for the float one.
If the values were defined using a random separator ('~'), any vector between (1, 2, 3) and (4.0, 5.0, 6.0) or any float value between 1.0 and 3.0 could have been returned.
Please also note that controlled randoms using lists works also with non-numerical values.

Lists can span multiple lines when using # at the end of the line (line ending comments are allowed).
If you want to define an empty element at the end of a list, use ##

Example:

```
Key2 = Var1 # Var2 #
       Var3 #; This list still continues on next line and this comment is
valid.
       Var4  ; This list is now complete and contains 4 elements.
Key3 = Var1 # Var2 ##; This list will not span on the next line but will
contain a 3rd (and last) empty element.
```

1)
cyclic inheritance is **not** supported
2)
Ints, Floats and Vectors
3)
In order to guarantee the generated value will be different on each run, call orxMath_InitRandom at the start of the program. Seed it with a perpetually changing integer, e.g. milliseconds since the Unix epoch.

From:
https://orx-project.org/wiki/ - **Orx Learning**

Permanent link:
https://orx-project.org/wiki/en/orx/config/syntax?rev=1496130646

Last update: **2025/09/30 17:26 (14 months ago)**