The Unofficial Guide to xcconfig files

Table of Contents

General Information

One of the least documented aspects of the configuration process are xcconfig files. As of this writing there seem to be no documents provided by Apple that explain how to use xcconfigs or why they exist. A xcconfig file is used as a supplemental file to a specific build configuration. A build configuration can have an associated xcconfig file, this allows for additional changes to the target's build settings from outside the Xcode project editor.

↑ Table of Contents

Syntax

xcconfig files follow a small set of syntax rules:

- Comments:
  - start with `//` and continue until the end of the line.
  - there is no support for multi-line comments.

- Include statements:
  - start with `#include` then have a path in double quotes. (See the section #include Statements for more details)

- Variables:
  - start with `_` or an upper or lowercase letter.
  - can contain the following characters:
    - underscore `_`
    - numbers `0` through `9`
    - letters `a` through `z` and `A` through `Z`

- Assignment:
  - is done by use of `=` between the variable name and the value.

- Lines:
  - each line is evaluated as a separate item, a single value cannot span multiple lines.
  - If a line ends in `;` then the semi-colon is ignored, this does not act as a line delimiter.

- Strings:
  - Can be represented with both double quotes `"` and single quotes `'`

If you violate any of the syntax rules, Xcode will ignore the xcconfig file entirely and display a warning describing the problem.

↑ Table of Contents

#include Statements

While Xcode limits the assignment of one xcconfig file per target per build configuration, you can import additional settings from other xcconfig files. To import the contents of another xcconfig file the line must start `#include` then followed by a file path that is inside of a set of double quotes.

```
#include "Debug.xcconfig"
#include"Shared.xcconfig"
```

Both of the lines above are valid syntax, there does not need to be a space separating the include statement from the path.

Xcode will search for file based on how the path starts:

- Starts with `/`
  - This dictates that the file path is an absolute path on the file system (`/` is the root volume).

Example:

```
#include "/Users/sam/Documents/shared.xcconfig" // includes settings fr
```

- Starts with any other characters
  - This dictates that the file path is relative to the location of that xcconfig file.

Example:

```
#include "default.xcconfig" // includes settings from the "default.xcco

#include "../OtherConfigs/Shared.xxconfig" // includes a config file fr
                                           // called "OtherConfigs"
```

- Starts with `<DEVELOPER_DIR>`
  - This is a special case that will allow you to specify a path based on the value of `$(DEVELOPER_DIR)`.

Example:

```
#include "<DEVELOPER_DIR>/Makefiles/CoreOS/Xcode/BSD.xcconfig"
```

↑ Table of Contents

Variable Assignment

Variables are assigned by placing an equals sign `=` after the variable name. Any amount of whitespace (both a regular space and a tab are valid whitespace) can exist between the variable name and the equals sign, as well between the equals sign and the value that is being assigned.

To use the default value for that variable, do not put anything after the equals sign. This will use the default for that variable or will not perform an assignment. This behavior should be used when wanting to select values that are under "Automatic" headings in the value drop-downs in the build settings.

Overriding

Build setting variables set on the Project or Target level can be overridden by reassigning the value of that variable in a xcconfig file.

```
// Variable set in the project file
OTHER_LDFLAGS = -ObjC

// lib.xcconfig
OTHER_LDFLAGS = -framework Security
```

When compiling with this, the `-ObjC` value is going to be overridden by the new value `-framework Security`.

For more information on how variable assignment priority works, please see the section Build Setting Inheritance.

Inherit

There is a special variable that can be used that will allow you to get the existing value of the variable so variable assignment isn't destructive.

```
// Variable set in the project file
OTHER_LDFLAGS = -ObjC

// lib.xcconfig
OTHER_LDFLAGS = $(inherited) -framework Security
```

When compiling with this, the value of `OTHER_LDFLAGS` is going to be `-ObjC -framework Security`.

For more information on how variable assignment priority works, please see the section Build Setting Inheritance.

↑ Table of Contents

Conditional Variable Assignment

In addition to regular variable assignment, you can have variables be assigned if a set of conditions are met. For example, changing the linker flags used based

on what OS version you are building for. The value check of the conditional supports the wildcard character `*` to perfom evaluation.

Something to take note of, the conditional assignment takes precedence over other assignments. For example:

```
FOO             = bar
FOO[sdk=macosx*] = buzz
```

If your target is building for OS X and iOS the build settings will look like this:

```
|-- FOO                =   bar
    |--   Any Mac SDK   =   buzz
    |--   Any iOS SDK   =   bar     // this is going to inherit from th
```

Multiple conditionals can be combined to create very specific variable assignments. For example:

```
FOO[sdk=macosx*][arch=i386] = bar // This only gets assigned if buildin
                                  // and targeting 32bit Intel architec
```

There are two style of multi-condition assignment:

```
FOO[sdk=<sdk>][arch=<arch>] = ...
```

and

```
FOO[sdk=<sdk>,arch=<arch>] = ...
```

There are 5 known conditional "flavours" that can be checked against: SDK, Architecture, Build Configuration name, Build Variant, and Dialect.

SDK

The sdk conditional assignment operates based on the value of `$(SDKROOT)`.

This is used to configure values to be assigned based on the selected SDK.

```
FOO[sdk=macosx10.8]         = ...   // For building against the 10.8 SD
FOO[sdk=macosx10.9]         = ...   // For building against the 10.9 SD
FOO[sdk=macosx10.10]        = ...   // For building against the 10.10 S
FOO[sdk=macosx*]            = ...   // For building against any Mac OS

FOO[sdk=iphoneos*]          = ...   // For building against any iOS SDK
FOO[sdk=iphonesimulator*]   = ...   // For building against any iOS Sim

FOO[sdk=*]                  = ...   // For building against any SDK
```

Arch

The arch conditional assignment operates based on the value of `$(CURRENT_ARCH)`. The `$(CURRENT_ARCH)` build setting comes from `$(ARCHS)`.

```
FOO[arch=i386]      = ...   // For building to target 32bit Intel
FOO[arch=x86_64]    = ...   // For building to target 64bit Intel

FOO[arch=armv7]     = ...   // For building to target ARM v7
FOO[arch=arm64]     = ...   // For building to target ARM64
FOO[arch=arm*]      = ...   // For building to target any ARM

FOO[arch=*]         = ...   // For building to target any architecture
```

Config

The configuration conditional assignment operates based on the value of `$(CONFIGURATION)`.

This will not behave as one might expect. Unlike using the `[sdk=]` or `[arch=]` conditional assignments, the `[config=]` will also implicitly add `[sdk=*][arch=*]` as conditions to satisfy for assignment. For example, the following line:

```
ONLY_ACTIVE_ARCH[config=Debug] = YES
```

really looks like:

```
ONLY_ACTIVE_ARCH[config=Debug][sdk=*][arch=*] = YES
```

While this doesn't look like it is a big deal, as the two additional conditions should fall through successfully and only apply the value to the "Debug" configuration -- this is not how Xcode resolves build setting assignment. What this will do instead is create a subset of the specific build configuration where the value will be assigned.

What you most likely want:

```
|-- VALUE
    |--   Debug     <YOUR DEBUG VALUE>
    |-- Release     <YOUR RELEASE VALUE>
```

What Xcode will see:

```
|-- VALUE
    |--   Debug                     <EMPTY OR DEFAULT VALUE>
        |-- Any SDK OR Any Arch     <YOUR DEBUG VALUE>
    |-- Release                     <EMPTY OR DEFAULT VALUE>
        |-- Any SDK OR Any Arch     <YOUR RELEASE VALUE>
```

This is significant because when performing a build from a scheme, it will use the assigned build configuration to query for the value to use for all the build settings. Since this won't match the parameters as defined by the subset value created from `[config=]`, it uses the top level value based on the configuration. This means whatever value is manually set there or the inherited default value from the next level up (Target/Project/SDK). Because of this, using `[config=]` should almost never be the desired behavior when assigning values based on build configuration name.

Please see the section Variable Substitution to see how to assign variables based on build configuration name.

Variant

The variant conditional assignement operates on the value of `$(CURRENT_VARIANT)`. The `$(CURRENT_VARIANT)` build setting comes from `$(BUILD_VARIANTS)`.

This option shouldn't be used.

Dialect

While this is a valid conditional check, it is unclear what it uses to evaluate.

This option shouldn't be used.

↑ Table of Contents

Variable Substitution

Variable assignment is not limited to values. You can reference the values of other variables to be used in assignment. There are two ways to reference another variable:

For variable named `FOO`:

- `$(FOO)`
- `${FOO}`

Both styles are interchangeable for all variables, including the special `$(inherited)` variable.

Example:

```
HELLO = hello
WORLD = world
FOO = $(HELLO) ${WORLD} // The value of FOO is "hello world"
```

Additionally, variable assignment can become more complex that the scope of the conditional assignment checks allow. To perform more complex conditional variable assignments you can use variable substitution.

For example, you have an application target and a unit test target. You want to change the version number based on if it is compiling the app or the unit test. Since the conditional assignment checks won't be able to look at those details for you that leaves two options. Either create separate xcconfig files for the application and the unit test targets, or use variable substitution to assign based on another variable.

Example:

```
CURRENT_PROJECT_VERSION_app = 15.3.9 // Application version number
CURRENT_PROJECT_VERSION_xctest = 1.0.0 // Unit Test version number

CURRENT_PROJECT_VERSION = $(CURRENT_PROJECT_VERSION_$(WRAPPER_EXTENSION
```

Xcode will assign this by first resolving the value of `$(WRAPPER_EXTENSION)`. For building an application the value of this variable will be `app` and for a unit test it will be `xctest`. This will then create a variable reference to either `$(CURRENT_PROJECT_VERSION_app)` or `$(CURRENT_PROJECT_VERSION_xctest)` and assign the respective value associated with that to `CURRENT_PROJECT_VERSION`. If you were to build this while `$(WRAPPER_EXTENSION)` was set to nothing, this would create the variable `$(CURRENT_PROJECT_VERSION_)`. Since there is no declaration of that

value, nothing would be assigned to override the value of `CURRENT_PROJECT_VERSION`.

This method of conditional assignment is very useful and powerful for organizing build settings. Please keep in mind that while the right side of the assignment operator can contain any type of character, the variable names themselves cannot. Please refer to the Syntax section for the specifications surrounding the valid characters that can be used in variable names.

NOTE: There is a way to work around the limitations of invalid character names. If you edit the `project.pbxproj` file in the Xcode project file and add new values to the relevant XCBuildConfiguration objects by enclosing the name in double quotes, the variables will register as valid and be displayed under the "User-Defined" settings section in the Xcode editor. While these settings are visible and can be used to substitute values elsewhere in the project or in the xcconfig files this is not supported behavior. Setting with names that contain invalid characters will not get properly exported to be used in other parts of the build system.

↑ Table of Contents

Build Setting Inheritance

> Disclaimer: This may contain some inaccuracies, behavior documented as reverse engineered from Xcode.

Xcode represents build settings as a set of "levels" that will have a single resolved value that is used for the build process. The value that each build setting variable has is inherited from the previous level.

Inheritance is performed in the following order (least to highest prescedence):

- Platform defaults
- Project file
- xcconfig file for the Project file
- Target
- xcconfig file for the Target

Value assignment is performed in the following order (least to highest prescedence):

- Platform defaults
- xcconfig for the Project file
- Project file

- xcconfig for the Target
- Target

The distinction between the ordering of these two operations is important to make note of. The difference in behavior can lead to some odd bugs in performing variable assignment.

Example: The Target on a Project file defines `PRODUCT_NAME` as `MyApp`. There is a xcconfig file assigned to this target that contains the following:

```
//
// Config.xcconfig
//

PRODUCT_NAME = testing
PRODUCT_NAME_ORIGINAL = $(PRODUCT_NAME) // The value of `PRODUCT_NAME_O
                                        // as assigned by the line befo
                                        // is "MyApp", because the inhe
                                        // over assignment.

// ...

FOO_MyApp = MyAppsName
FOO_testing = MyAppsNewName
BAR = $(FOO_$(PRODUCT_NAME))            // This will also use the value
                                        // and resolve to be "$(FOO_MyA
```

Inheritance only works between levels, variable assignments perform on the same level will override the previous assignment. You cannot use `$(inherited)` in an xcconfig file to get the value assigned for a variable from an imported xcconfig file. To use that value you must use separate names and reference those variables in assignment.

↑ Table of Contents

Resources

- Legacy Xcode Build System Reference (mirror)

↑ Table of Contents
