FASTBUILD

Syntax Guide

Overview

Details of the FASTBuild configuration file syntax can be grouped into 4 categories:

Formatting

- Whitespace
- Comments
- Quotation
- Escaping

Directives

- #define / #undef
- #if / #endif
- #import
- #include
- #once

Variables

- Declaration and Types
- Modification
- Scoping

- Structs
- Dynamic Construction

Functions

- Calling
- Properties
- Arguments
- Build-Time Substitutions

Formatting

Whitespace

Whitespace (i.e. spaces, tabs, carriage returns and linefeeds) are all ignored during parsing and have no syntactic significance.

Comments

Comments can occur anywhere in the file. All characters to the end of the current line are ignored when a comment character is encountered. Comments can be started with a double forward slash.

```
// A comment
```

Or a semi-colon:

```
; A comment
```

Quotation

Strings are quoted with the " or ' character. Two quotation characters are supported to allow strings which contains quotes. For example

```
"This contains ' inside the string"
'This contains " inside the string'
```

Alternatively, strings can contains quotes through the use of escaping.

Escaping

Characters that have special meaning within strings, such as ", ' or $ can be escaped with the ^ character as follows:

```
"This string has ^" in it"
'This string has ^' in it'
"Escape the ^$ and ^" symbols, or escape the ^^ itself."
```

Directives

#define / #undef

The #define / #undef directives allow user defined tokens to simplify BFF control flow.

Examples:

```
#if __WINDOWS__
    #define ENABLE_PDB_STORING
#endif
#if ENABLE_PDB_STORING
    ...
```

```
#define USE_CLANG_COMPILER
#if __WINDOWS__
    #undef USE_CLANG_COMPILER
#endif
#if USE_CLANG_COMPILER
    ...
```

#if / #endif

The #if / #endif directive pair allows parts of a bff configuration file to be applied conditionally, depending on previous #define declarations. The #if directive can be negated using the ! operator.

Examples:

```
#if __WINDOWS__
        .CompilerOptions = '/c "%1" /Fo"%2"'
#endif
#if !__WINDOWS__
        .CompilerOptions = '-c "%1" -o "%2"'
#endif
```

The following pre-defined (and reserved) symbols are available for use:

Symbol	Value
__LINUX__	Defined if running on Linux
__OSX__	Defined if running on OS X
__WINDOWS__	Defined if running on Windows

Additionally, the "exists" operator can be used to branch on the existence of an environment variable.

Example:

```
#if exists(MY_ENV_VAR)
    #import MY_ENV_VAR
#endif
```

#import

The #import directive allows a bff configuration file to import an Environment Variable. A variable with the same name as the Environment Variable will be created at the scope of the #import, as if it was locally declared by a normal variable assignment.

Examples:

```
#import VS120COMNTOOLS
.VisualStudio2012Compiler = '$VS120COMNTOOLS$\..\..\VC\bin\cl.exe'
```

#import can be combined with #if exists to support importation of optional environment variables.

#include

The #include directive allows a bff configuration file to include another. This allows the configuration to be split along logical lines, such as per-platform or per-configuration.

Examples:

```
#include "platforms/x86.bff"
#include "platforms/x64.bff"
```

```
#include "libs/core.bff"
#include "libs/graphics.bff"
#include "libs/sound.bff"

#include "game.bff"
```

#once

The #once directive specifies that a bff file should only be parsed once, regardless of how many times it is included.

Examples:

```
// Common.bff - containing common configuration options
#once
```

```
// LibraryA.bff
#include "Common.bff"
```

```
// LibraryB.bff
#include "Common.bff"
```

```
// fbuild.bff - the root config file
#include "LibraryA.bff"
#include "LibraryB.bff"
```

Variables

Declaration and Type

Variable declarations begin with a period followed by a contiguous block of alphanumeric characters. Variable names are case insensitive. Four variable types are supported (String, Integer, Boolean and Array). Variable types are implied by their declaration.

Examples:

```
.MyString = "hello"
.MyBool   = true
.MyInt    = 7
.MyArray  = { "aaa", "bbb", "ccc" }
```

Modification

Variables can be overridden or modified at any point after they have been declared, as follows:

```
; Declaration
.MyString = "hello"

; Concatenation
.MyString + " there"
          + " FASTBuild user" ; Uses last referenced variable automatically

; Subtraction
.MyString - " user"
          - "hello "; Uses last referenced variable automatically

; Re-assignment
.MyString = "hello"
```

Variables can be constructed from other variables:

```
.StringA = "hello"
.StringB = "$StringA$ FASTBuild user!"
```

Scoping

Variables are valid for the scope they are declared in, and all sub-scopes. Modifications to variables within a scope exist only in the scope they are made. For example:

```
.MyString = 'hello'
{
  .MyString = 'goodbye'
}
; Mystring contains 'hello' again

{
  .MyString + ' hello'  ; string contains 'hello hello'
}
; Mystring contains 'hello' again
```

Variables can be pushed to the parent last used parent scope using the ^ prefix:

```
.CompilerOptions     = '/c "%1" -o "%2"'

.IncPaths            = { 'libA/inc/', 'libB/inc/' }
ForEach( .IncPath in .IncPaths )
{
  ^CompilerOptions   + ' /I"$IncPath$"'
}
```

Structs

Structs allow the grouping of variables to allow reuse of settings in complex build configurations. Structs contain any number of variables of any type (including other structs). Arrays of structs are also permitted.

```
.MyStruct =
[
  .MyString = "string"
  .MyInt    = 7
]
```

A struct contains a number of properties, isolating them from the current namespace. The Using function allows all the members of a struct to be pushed into the current scope:

```
Using( .MyStruct )
```

The Using function also allows Structs to effectively inherit and override other structs.

```
.StructA = [ .StringA = 'a' ]
.StructB =
[
  Using( .StructA ) // "inherit" - StructB now has a StringA property
  .StringB = 'b'    // "extend" - An additional property
]
```

Structs can be concatenated using the plus operator. The plus operator will be applied recursively to each member within the struct.

```
.StructA =
[
  .ArrayOfStrings     = { 'String1', 'String2' }
  .String             = 'String'
]
```

```
.StructB =
[
  .ArrayOfStrings     = { 'String3' }
  .String             = '!'
]
.StructC = .StructA
         + .StructB
// StructC now contains .ArrayOfStrings = { 'String1', 'String2', 'String3' } and .String =
```

Looping through arrays of structures is a useful technique for minimizing configuration complexity.

```
.ConfigX86 =
[
  .Compiler   = "compilers/x86/cl.exe"
  .ConfigName = "x86"
]
.ConfigX64 =
[
  .Compiler = "compilers/x64/cl.exe"
  .ConfigName = "x64"
]
.Configs = { .ConfigX86, .ConfigX64 }

ForEach( .Config in .Configs )
{
  Using( .Config )
  Library( "Util-$ConfigName$" )
  {
    .CompilerInputPath  = 'libs/util/'
    .CompilerOutputPath = 'out/$ConfigName$/'
    .LibrarianOutput    = 'out/$ConfigName$/Util.lib'
  }
}
```

Dynamic Construction

Variable names can be dynamically constructed at parse time as follows:

```
.Config   = 'Debug'
.Platform = 'X64'
.Options  = ."CompilerOptions_$Config$_$Platform$" // expands to .CompilerOptions_Debug_X64
```

Functions

NOTE: See the Function Reference for a detailed list of all available functions and their Arguments and Properties.

Calling

Functions describe the dependency information to FASTBuild. Each library, executable, unit test or other "node" is described to FASTBuild through the use of Functions.

Functions generally take the form:

```
Function( args )
{
  .Property1 = "value1"
  .Property2 = "value2"

  ; etc...
}
```

Properties

Functions are primarily controlled by their Properties, taken from active variable declarations (either internally declared or from an inherited scope). The recognized properties vary from function to function.

The result of the following two examples are equivalent:

```
; Example A
Library( "mylib" )
{
  .Compiler = "cl.exe"
}

; Example B
.Compiler = "cl.exe"
Library( "mylib" )
{
}
```

Variables interact with Function properties in this way to allow common declarations to be moved to higher level scope to avoid duplication. Combined with the variable scoping rules, this also allows for bespoke specializations.

Arguments

Arguments are often optional, and their quantity and syntax varies from function to function. Where they are not required (or optional), brackets should be omitted.

As an example, the Library function takes an optional alias argument to allow a user-friendly name when targetting the library to be built on the command line:

```
Library( "mylib" )                          ; can target "mylib"...
{
  .LibrarianOutput = "out\libs\mylib.lib" ; ...instead of this
}
```

Build-Time Substitutions

Build-Time Substitutions allow FASTBuild to replace certain configuration values at build time, as opposed to configuration parsing time ($ tokens). This functionality is used whenever the value of an argument can't be known at configuration time, or there is a many to one relationship between the configuration and the number of nodes it results in during the build.

One common example of where build time substitutions are required is when building a library. Since the compiler is invoked for each file, but we only define one set of 'CompilerOptions', a build-time substitution is required. For example:

```
Library( "mylib" )
{
  ; NOTE: some option omitted for brevity
  .CompilerInputPath = 'Code\'               ; will build all cpp files in this directory
  .CompilerOutputPath= 'Tmp\'
  .CompilerOptions   = '%1 /Fo%2 /nologo /c' ; substitutions detailed below
}
```

Assuming there are 3 files in the 'Code' directory ('a.cpp', 'b.cpp' and 'c.cpp'), the compiler will be invoked 3 times. In each case %1 and %2 will be replaced with appropriate values for the input and output. In this case, that means 'a.cpp' & 'Tmp\a.obj' for the first invocation, 'b.cpp' & 'Tmp\b.obj' for the second, and finally 'c.cpp' & 'Tmp\c.obj'.

The behaviour of build-time substitutions is different for each Function (what values they provide and what properties they can be used on). For details, see the Function Reference.