NAME

cmake-language − CMake Language Reference

ORGANIZATION

CMake input files are written in the "CMake Language" in source files named **CMakeLists.txt** or ending in a **.cmake** file name extension.

CMake Language source files in a project are organized into:

- *Directories* (**CMakeLists.txt**),
- *Scripts* (**<script>.cmake**), and
- *Modules* (**<module>.cmake**).

Directories

When CMake processes a project source tree, the entry point is a source file called **CMakeLists.txt** in the top−level source directory. This file may contain the entire build specification or use the **add_subdirectory()** command to add subdirectories to the build. Each subdirectory added by the command must also contain a **CMakeLists.txt** file as the entry point to that directory. For each source directory whose **CMakeLists.txt** file is processed CMake generates a corresponding directory in the build tree to act as the default working and output directory.

Scripts

An individual **<script>.cmake** source file may be processed in *script mode* by using the **cmake(1)** command−line tool with the **−P** option. Script mode simply runs the commands in the given CMake Language source file and does not generate a build system. It does not allow CMake commands that define build targets or actions.

Modules

CMake Language code in either *Directories* or *Scripts* may use the **include()** command to load a **<module>.cmake** source file in the scope of the including context. See the**cmak e−modules(7)** manual page for documentation of modules included with the CMake distribution. Project source trees may also provide their own modules and specify their location(s) in the **CMAKE_MODULE_PATH** variable.

SYNTAX

Encoding

A CMake Language source file may be written in 7−bit ASCII text for maximum portability across all supported platforms. Newlines may be encoded as either **\n** or **\r\n** but will be converted to **\n** as input files are read.

Note that the implementation is 8−bit clean so source files may be encoded as UTF−8 on platforms with system APIs supporting this encoding. In addition, CMake 3.2 and above support source files encoded in UTF−8 on Windows (using UTF−16 to call system APIs). Furthermore, CMake 3.0 and above allow a leading UTF−8 *Byte−Order Mark* in source files.

Source Files

A CMake Language source file consists of zero or more *Command Invocations* separated by newlines and optionally spaces and *Comments*:

file ::= *file_element**
file_element ::= *command_invocation line_ending* |
(*bracket_comment*|*space*)* *line_ending*
line_ending ::= *line_comment*? *newline*
space ::= <match '[\t]+'>
newline ::= <match '\n'>

Note that any source file line not inside *Command Arguments* or a *Bracket Comment* can end in a *Line Comment*.

Command Invocations

A *command invocation* is a name followed by paren−enclosed arguments separated by whitespace:

command_invocation ::= *space* identifier space** '(' *arguments* ')'
identifier ::= <match '[A−Za−z_][A−Za−z0−9_]*'>
arguments ::= *argument*? *separated_arguments**
separated_arguments ::= *separation+ argument*? |
*separation** '(' *arguments* ')'
separation ::= *space* | *line_ending*

For example:

```
add_executable(hello world.c)
```

Command names are case−insensitive. Nested unquoted parentheses in the arguments must balance. Each **(** or **)** is given to the command invocation as a literal *Unquoted Argument*. This may be used in calls to the **if()** command to enclose conditions. For example:

```
if(FALSE AND (FALSE OR TRUE)) # evaluates to FALSE
```

NOTE:

CMake versions prior to 3.0 require command name identifiers to be at least 2 characters.

CMake versions prior to 2.8.12 silently accept an *Unquoted Argument* or a *Quoted Argument* immediately following a *Quoted Argument* and not separated by any whitespace. For compatibility, CMake 2.8.12 and higher accept such code but produce a warning.

Command Arguments

There are three types of arguments within *Command Invocations*:

argument ::= *bracket_argument* | *quoted_argument* | *unquoted_argument*

Bracket Argument

A *bracket argument*, inspired by *Lua* long bracket syntax, encloses content between opening and closing "brackets" of the same length:

bracket_argument ::= *bracket_open bracket_content bracket_close*
bracket_open ::= '[' '='* '['
bracket_content ::= <any text not containing a *bracket_close* with
the same number of '=' as the *bracket_open*>
bracket_close ::= ']' '='* ']'

An opening bracket is written **[** followed by zero or more **=** followed by **[**. The corresponding closing bracket is written **]** followed by the same number of **=** followed by **]**. Brackets do not nest. A unique length may always be chosen for the opening and closing brackets to contain closing brackets of other lengths.

Bracket argument content consists of all text between the opening and closing brackets, except that one newline immediately following the opening bracket, if any, is ignored. No evaluation of the enclosed content, such as *Escape Sequences* or *Variable References*, is performed. A bracket argument is always given to the command invocation as exactly one argument.

For example:

```
message([=[
This is the first line in a bracket argument with bracket length 1.
No \-escape sequences or ${variable} references are evaluated.
This is always one argument even though it contains a ; character.
The text does not end on a closing bracket of length 0 like ]].
It does end in a closing bracket of length 1.
]=])
```

NOTE:
CMake versions prior to 3.0 do not support bracket arguments. They interpret the opening bracket as the start of an *Unquoted Argument*.

Quoted Argument

A *quoted argument* encloses content between opening and closing double-quote characters:

quoted_argument ::= "" *quoted_element** ""
quoted_element ::= <any character except '\' or ""> |
escape_sequence |
quoted_continuation
quoted_continuation ::= '\' *newline*

Quoted argument content consists of all text between opening and closing quotes. Both *Escape Sequences* and *Variable References* are evaluated. A quoted argument is always given to the command invocation as exactly one argument.

For example:

```
message("This is a quoted argument containing multiple lines.
This is always one argument even though it contains a ; character.
Both \\-escape sequences and ${variable} references are evaluated.
The text does not end on an escaped double-quote like \".
It does end in an unescaped double quote.
")
```

The final \ on any line ending in an odd number of backslashes is treated as a line continuation and ignored along with the immediately following newline character. For example:

```
message("\
This is the first line of a quoted argument. \
In fact it is the only line but since it is long \
the source code uses line continuation.\
")
```

NOTE:
CMake versions prior to 3.0 do not support continuation with \. They report errors in quoted arguments containing lines ending in an odd number of \ characters.

Unquoted Argument

An *unquoted argument* is not enclosed by any quoting syntax. It may not contain any whitespace, **(**, **)**, **#**, **"**, or \ except when escaped by a backslash:

unquoted_argument ::= *unquoted_element*+ | *unquoted_legacy*
unquoted_element ::= <any character except whitespace or one of '()#"\'> |
escape_sequence
unquoted_legacy ::= <see note in text>

Unquoted argument content consists of all text in a contiguous block of allowed or escaped characters. Both *Escape Sequences* and *Variable References* are evaluated. The resulting value is divided in the same way *Lists* divide into elements. Each non−empty element is given to the command invocation as an argument. Therefore an unquoted argument may be given to a command invocation as zero or more arguments.

For example:

```
foreach(arg
    NoSpace
    Escaped\ Space
    This;Divides;Into;Five;Arguments
    Escaped\;Semicolon
    )
  message("${arg}")
endforeach()
```

NOTE:

To support legacy CMake code, unquoted arguments may also contain double−quoted strings (**"..."**, possibly enclosing horizontal whitespace), and make−style variable references (**$(MAKEVAR)**).

Unescaped double−quotes must balance, may not appear at the beginning of an unquoted argument, and are treated as part of the content. For example, the unquoted arguments **−Da="b c"**, **−Da=$(v)**, and **a" "b"c"d** are each interpreted literally. They may instead be written as quoted arguments **"−Da=\"b c\""**, **"−Da=$(v)"**, and **"a\" \"b\"c\"d"**, respectively.

Make−style references are treated literally as part of the content and do not undergo variable expansion. They are treated as part of a single argument (rather than as separate **$**, **(**, **MAKEVAR**, and **)** arguments).

The above "unquoted_legacy" production represents such arguments. We do not recommend using legacy unquoted arguments in new code. Instead use a *Quoted Argument* or a *Bracket Argument* to represent the content.

Escape Sequences

An *escape sequence* is a \ followed by one character:

escape_sequence ::= *escape_identity* | *escape_encoded* | *escape_semicolon*
escape_identity ::= '\' <match '[^A−Za−z0−9;]'>
escape_encoded ::= '\t' | '\r' | '\n'
escape_semicolon ::= '\;'

A \ followed by a non−alphanumeric character simply encodes the literal character without interpreting it as syntax. A**\t**, **\r**, or **\n** encodes a tab, carriage return, or ne wline character, respectively. A **\;** outside of any *Variable References* encodes itself but may be used in an *Unquoted Argument* to encode the **;** without dividing the argument value on it. A **\;** inside *Variable References* encodes the literal **;** character. (See also policy **CMP0053** documentation for historical considerations.)

Variable References

A *variable reference* has the form **${<variable>}** and is evaluated inside a *Quoted Argument* or an *Unquoted Argument*. A variable reference is replaced by the value of the variable, or by the empty string if the variable is not set. Variable references can nest and are evaluated from the inside out, e.g. **${outer_${inner_variable}_variable}**.

Literal variable references may consist of alphanumeric characters, the characters **/_.+−**, and *Escape Sequences*. Nested references may be used to evaluate variables of any name. See also policy **CMP0053**

documentation for historical considerations and reasons why the **$** is also technically permitted but is discouraged.

The *Variables* section documents the scope of variable names and how their values are set.

An *environment variable reference* has the form **$ENV{<variable>}**. See the*En vironment Variables* section for more information.

A *cache variable reference* has the form **$CACHE{<variable>}**. See**CA CHE** for more information.

The **if()** command has a special condition syntax that allows for variable references in the short form **<variable>** instead of **${<variable>}**. However, environment and cache variables always need to be referenced as **$ENV{<variable>}** or **$CACHE{<variable>}**.

Comments

A comment starts with a **#** character that is not inside a *Bracket Argument*, *Quoted Argument*, or escaped with \ as part of an *Unquoted Argument*. There are two types of comments: a *Bracket Comment* and a *Line Comment*.

Bracket Comment

A **#** immediately followed by a *bracket_open* forms a *bracket comment* consisting of the entire bracket enclosure:

bracket_comment ::= '#' *bracket_argument*

For example:

```
#[[This is a bracket comment.
It runs until the close bracket.]]
message("First Argument\n" #[[Bracket Comment]] "Second Argument")
```

NOTE:

CMake versions prior to 3.0 do not support bracket comments. They interpret the opening **#** as the start of a *Line Comment*.

Line Comment

A **#** not immediately followed by a *bracket_open* forms a *line comment* that runs until the end of the line:

line_comment ::= '#' <any text not starting in a *bracket_open*
and not containing a *newline*>

For example:

```
# This is a line comment.
message("First Argument\n" # This is a line comment :)
        "Second Argument") # This is a line comment.
```

CONTROL STRUCTURES

Conditional Blocks

The **if()**/**elseif()**/**else()**/**endif()** commands delimit code blocks to be executed conditionally.

Loops

The **foreach()**/**endforeach()** and **while()**/**endwhile()** commands delimit code blocks to be executed in a loop. Inside such blocks the**br eak()** command may be used to terminate the loop early whereas the **continue()** command may be used to start with the next iteration immediately.

Command Definitions
The **macro()**/**endmacro()**, and **function()**/**endfunction()** commands delimit code blocks to be recorded for later invocation as commands.

VARIABLES

Variables are the basic unit of storage in the CMake Language. Their values are always of string type, though some commands may interpret the strings as values of other types. The **set()** and **unset()** commands explicitly set or unset a variable, but other commands have semantics that modify variables as well. Variable names are case−sensitive and may consist of almost any text, but we recommend sticking to names consisting only of alphanumeric characters plus **_** and **−**.

Variables have dynamic scope. Each variable "set" or "unset" creates a binding in the current scope:

Function Scope
Command Definitions created by the **function()** command create commands that, when invoked, process the recorded commands in a new variable binding scope. A variable "set" or "unset" binds in this scope and is visible for the current function and any nested calls within it, but not after the function returns.

Directory Scope
Each of the *Directories* in a source tree has its own variable bindings. Before processing the **CMakeLists.txt** file for a directory, CMake copies all variable bindings currently defined in the parent directory, if any, to initialize the new directory scope. CMake *Scripts*, when processed with **cmake −P**, bind variables in one "directory" scope.

A variable "set" or "unset" not inside a function call binds to the current directory scope.

Persistent Cache
CMake stores a separate set of "cache" variables, or "cache entries", whose values persist across multiple runs within a project build tree. Cache entries have an isolated binding scope modified only by explicit request, such as by the **CACHE** option of the **set()** and **unset()** commands.

When evaluating *Variable References*, CMake first searches the function call stack, if any, for a binding and then falls back to the binding in the current directory scope, if any. If a "set" binding is found, its value is used. If an "unset" binding is found, or no binding is found, CMake then searches for a cache entry. If a cache entry is found, its value is used. Otherwise, the variable reference evaluates to an empty string. The **$CACHE{VAR}** syntax can be used to do direct cache entry lookups.

The **cmake−variables(7)** manual documents the many variables that are provided by CMake or have meaning to CMake when set by project code.

NOTE:
CMake reserves identifiers that:

- begin with **CMAKE_** (upper−, lower−, or mixed−case), or
- begin with **_CMAKE_** (upper−, lower−, or mixed−case), or
- begin with **_** followed by the name of any **CMake Command**.

ENVIRONMENT VARIABLES

Environment Variables are like ordinary *Variables*, with the following differences:

Scope Environment variables have global scope in a CMake process. They are never cached.

References
Variable References have the form **$ENV{<variable>}**.

Initialization
Initial values of the CMake environment variables are those of the calling process. Values can be changed using the **set()** and **unset()** commands. These commands only affect the running CMake

process, not the system environment at large. Changed values are not written back to the calling process, and they are not seen by subsequent build or test processes.

The **cmake−env−variables(7)** manual documents environment variables that have special meaning to CMake.

LISTS

Although all values in CMake are stored as strings, a string may be treated as a list in certain contexts, such as during evaluation of an *Unquoted Argument*. In such contexts, a string is divided into list elements by splitting on **;** characters not following an unequal number of **[** and **]** characters and not immediately preceded by a \. The sequence**\;** does not di vide a value but is replaced by **;** in the resulting element.

A list of elements is represented as a string by concatenating the elements separated by **;**. For example, the **set()** command stores multiple values into the destination variable as a list:

```
set(srcs a.c b.c c.c) # sets "srcs" to "a.c;b.c;c.c"
```

Lists are meant for simple use cases such as a list of source files and should not be used for complex data processing tasks. Most commands that construct lists do not escape **;** characters in list elements, thus flattening nested lists:

```
set(x a "b;c") # sets "x" to "a;b;c", not "a;b\;c"
```

COPYRIGHT