CoreTML framework 1.4

Template Markup Language Reference

http://coretml.sourceforge.net

1. Introduction

Template Markup Language (TML) is designed to make it possible to insert special content (such as control sequences) into any non-binary files (e.g. plain text files, source code in any programming language, documents written in other markup languages such as HTML, RTF or TeX). That special content is then processed when the file including such content is fed to an appropriate piece of software (e.g. TMLgen which is a part of the CoreTML framework).

The special content is organized with the help of a tagging system which ensures that control sequences (which must be processed by a TML interpreter) can always be told apart from the user-defined text (which is irrelevant for TML interpreter).

CoreTML framework uses two flavours of TML:

- **TML-g** – the main language, which is designed to create parametrized templates that can be used to generate text files (e.g. source code) given the appropriate parameters.
- **TML-c –** the supplementary language with similar syntax that is used to describe graphical user interface (GUI) for setting up parameters for TML-g tempates.

These language flavours have an identical tagging system (described below) but define different tags with different semantics. Other TML flavours can also be created.

2. Basic syntax features

The tagging system is the same for all TML flavours. For user-defined text, the following rules apply:

- Any backslash character (`'\'`) should be preceded by an another backslash, i.e. represented as `'\\'`.
- Any closing brace character (`'}'`) should be also preceded by a backslash.
- Any opening brace character (`'{'`) may be (optionally) preceded by a backslash.

Tags are represented as follows:

```
\tag_name {content}
```

tag_name is a tag identifier which can include only alphanumeric characters or an underscore character (`'_'`). Tag identifier is always preceded by a backslash. *content* is any text between braces, its meaning being defined by a *tag_name*. Any number of whitespaces, tabulation characters and newline characters is allowed after *tag_name* and before the opening brace.

3. TML-g semantics

TML-g is designed to create templates. Normally, a template would then be fed to the TML-g interpreter (such as TMLgen) along with the appropriate parameters, and it would generate an output file based on the template. Usually, but not absolutely necessarily, TML-g templates would be used to create a kind of formal texts (e.g. source code in some programming language or hardware description language).

The result of tag processing is placed into the *output stream*. For top-level tags, this means writing to the output file (or to the standard output if an output file hasn't been specified). For nested tags, the resulting data are passed to the parent tag which interprets them as its content (parent tag doesn't differentiate between a plain text and a text created by nested tags).

The way of dealing with tag *content* is fully determined by a tag identidier. Basically, there are four methods of interpreting tag content:

1. *freetext* – content is considered to be some user-defined text (optionally containing nested tags). Formatting rules described above are still in force.

2. *string* – content is considered to be a string containing some information for the interpreter, which is not inserted directly to the output stream. The tag content can be surrounded by singular or double quotes, which are ignored. Any invisible characters (whitespaces, tabulations, newlines) at the beginning and at the end of the content are also ignored.

3. *argument list* – content is considered to be a set of variable assignments that can be separated with newlines and/or semicolons:

```
variable1=value1

variable2=value2;variable3=value3

...
```

   Variable names depend on the corresponding tag, and values must be valid Lua expressions.

4. *subtags* – some tags require its content to be comprised of nested tags.

TML-g makes extensive use of Lua scripting language. Lua itself isn't described in this reference. See http://www.lua.org for detailed Lua documentation.

As a special exception, if the first line of the template begins with #! (“shebang”), this line is skipped by an interpreter. If you need to insert a shebang at the start of the output stream, you can use `\x{g}` instead (see also `\x` tag description).

3.1. Basic tags

\comment

(content type: *freetext*)

A comment.

Examples

```
\comment{This text doesn't affect anything}
\comment{
     \echo{This is also not processed}
}
```

Description

The content of this tag is ignored.

\x

(content type: *string*)

Inserts special character or characters. Its content should consist of the following sequences:

- `n` – denotes an end-of-line character (inserts 0D0A or 0A depending on platform);
- `t` – a tabulation character;
- `s` – a whitespace;
- `g` – a shebang (`#!`) that can be also used at the start of the template (otherwise you can use `#!` characters directly);
- hexadecimal numbers (two-digit or one-digit) – denotes characters with corresponding character codes.

Examples

Insert one end-of-line character:

```
\x{n}
```

Insert three end-of-line characters:

```
\x{nnn}
```

Insert three characters: tabulation character and characters with codes 0E and 0F:

```
\x{t0E0F}
```

Insert characters with code 0E and 01.

```
\x{0E1}
```

\echo

(content type: *freetext*)

Writes content to the standard error stream (stderr), e.g. prints them in the console window.

Examples

```
\echo{Starting processing...}
```

\include

(content type: *string*)

Includes another TML-g file. The interpreter behaves exactly as if that file's contents were inserted in place of this tag, with the exception of strict formatting mode behavior, which must be set for each TML-g file separately.

The file name can include absolute or relative path. See “Paths and directories” chapter on how paths are processed in TML-g.

Examples

```
\include{scriptlib.tml}
```

\includetext

(content type: *string*)

Includes a plain text file, that is, transfers the content of that file to the output stream. The file itself isn't processed by the interpreter, so it doesn't have to adhere to TML tagging syntax.

The file name can include absolute or relative path. See “Paths and directories” chapter on how paths are processed in TML-g.

Examples

`\includetext` tag can be used to include Lua scripts from external files which don't need to adhere to TML tagging syntax:

```
\script{\includetext{exscript.lua}}
```

\silent

(content type: *freetext*)

The content of the *silent* tag is processed by the interpreter, but nothing is written to the output stream. Other actions (script executing etc.) are performed as usual.

Examples

```
\silent{
     foo bar\x{n}
     \echo{test}
}
```

In this example the words “foo bar” as well as newline characters won't be placed to the output stream, but the word “test” will still be written to the standard error stream.

\format

(content type: *argument list*)

This tag is used to control the behavior of the interpreter. One or more of the following variables can be set:

- *indent* (string). Controls output indentation. If *indent* is set to a non-empty string, that string will be inserted at the beginning of every single line.
- *indent+* (string). Appends its value to the current indentation string.
- *indent-* (number). Removes a specified number of characters from the end of the indentation string.
- *strict* (boolean). Activates or deactives strict formatting mode. When strict formatting mode is on, the following rules apply:
  - Any newline characters in the user-defined text are ignored. Newlines must be inserted explicitly with the \x tag.
  - Any spaces or tabulation characters in the template file at the beginning of every single line are ignored. Indentation must be controlled via *indent* parameter (see above) or with \x tag.

  This mode is designed to maintain reasonable text formatting in the output file while being able to insert newlines or indents in the template without having to worry that they will corrupt the output formatting. It is off by default.
- *once* (boolean). This mode ensures that the current template can be included only once in any other template. When using this mode, place `\format{once=true}` at the very beginning of the template.

There is also one special type of format tag content that resets all formatting variables to their default values:

```
\format{clear}
```

Examples

Set indentation string to “\t” (one tabulation character):

```
\format{indent="\t"}
```

Add “>>” to indentation string (it becomes “\t>>”):

```
\format{indent+=">>"}
```

Remove one last character from the indentation string (it becomes “\t>”):

```
\format{indent-=1}
```

Activate *strict* and *once* modes:

```
\format{strict=true;once=true}
```

\exit

(content type: *freetext*)

Prints the content on the standard error stream (if the content is non-empty) and exit.

Examples

Exit without printing anything:

```
\exit{}
```

Print message and exit:

```
\exit{Exiting from template...}
```

\error

(content type: *freetext*)

Generates an unconditional error, writes the tag's content to the standard error stream. The interpreter exits immediately after processing this tag. This tag differs from the previous one in that in this case the interpreter reports an error.

Examples

```
\error{One or more parameters have invalid values}
```

\assert

(content type: *freetext*)

Evaluates content as Lua expression and generate an error if the expression value is *false*, 0, “” (empty string) or *nil*.

Examples

Exit if *n* is lesser or equal to 0:

```
\assert{n>0}
```

3.2. Scripting tags

\script

(content type: *freetext*)

Executes a Lua script. Lua programming language will not be described in this manual; refer to the Lua website (http://www.lua.org) for Lua documentation.

If a Lua code returns a value that has *string* or *number* type, that value will be placed to the output stream. To write to the output stream directly from Lua, use *write()* function. See also the "Lua extensions" chapter for the list of additional Lua functions available in TML-g.

TML-g uses the same Lua context for the whole template, that is, variables and functions declared in one tag will be available in the others.

Note that TML tagging syntax continues to apply inside `\script` tag. For example, to print a backslash character in Lua you would normally write:

```
print("\\")
```

But TML syntax also requires that any backslash should be doubled, so under in TML-g you should write:

```
\script{print("\\\\")}
```

You can also place a Lua script in a separate file and invoke

```
\script{\includetext{filename.lua}}
```

In this case TML tagging syntax doesn't apply to the Lua script.

Examples

Euclidean algorithm for finding the greatest common denominator (GCD):

```
\script{
     a=x
     b=y
     write("GCD of "..x.." and "..y.." is: ")
     while a~=b do
          if a>b then a=a-b else b=b-a end
     end
     return a
}
```

Create an identity matrix of size n:

```
     \script{
          A=\{\}    -- declare a table; note the
                    -- backslashes  which are part
                    -- of the TML (not Lua) syntax
          for r=1,n do
               A[r]=\{\} -- create row
               for c=1,n do
                    A[r][c]=0
```

```
                end
                A[r][r]=1
            end
    }
```

\eval

(content type: *freetext*)

Evaluates content as a Lua expression; if it has *string* or *number* type, places its value into the output stream.

Examples

The following example writes 4.5 and 27 to the output stream:

```
\script{a=4.5;b=6}
\eval{a}
\eval{a*b}
```

3.3. Tags related to template invocation

\snippet

(content type: *subtags*)

Defines a snippet (a reusable piece of TML code). Just like templates, snippets can assume parameters as inputs and can be later invoked with the use of a `\create` tag. `\snippet` tag must contain two nested tags: `\name` and `\body`.

\name

(content type: *string*)

Valid only inside the `\snippet` tag content. Defines a name (identifier) of a snippet.

\body

(content type: *freetext*)

Valid only inside the `\snippet` tag content. Defines a snippet body.

Examples

Example of snippet definition (*name*, *surname*, *birth* and *occupation* are the assumed parameters in this example):

```
    \snippet{
        \name{namerecord}
        \body{
            First name: \eval{name}
            Last name: \eval{surname}
            Date of birth: \eval{birth}
```

```
                Occupation: \eval{occupation}
            }
        }
```

For an example on how to invoke a snippet, see examples for `\create` tag.

\create

(content type: *argument list*)

A powerful tag that is used to invoke templates and snippets. It can have any number of arguments that are passed directly to the template or snippet being invoked, but some of them also have special meaning:

- *template* (string). Template file name or path.
- *snippet* (string). Snippet name.
- *output* (string). Output file name or path.
- *outputdir* (string). Output directory.
    - If used along with the *output* argument, and the output file path is relative, the output file path is constructed relative to *outputdir.*
    - If used without an *output* argument or *output* specifies an absolute path, it doesn't influence the output of the spawned template.
    - In any case, *outputdir* is used as a base output directory for the spawned template, i.e. it is used as default output path for other templates that are spawned from that template. See also the *Paths and directories* chapter.

Either *template* or *snippet* arguments must be provided (but not both of them) to point out the source. Providing *output* parameter is optional: if it is not set, the processing results are written to the current output stream (instead of creating a new file). This provides a developer with maximum flexibility: both templates and snippets can be instantiated both “inside” and “outside” the current output stream.

Invoked templates and snippets are processed in a brand new context: the only link between the calling and called contexts is a set of user-defined parameters that are passed via a `\create` tag. Variables, Lua functions and other objects of the calling template are unavailable to the called one and vice versa.

The parameters passed via a `\create` tag must have one of the following Lua types: number, string, boolean, table or nil.

The file name can include absolute or relative path. TML-g interpreter will create output directories automatically if needed. See “Paths and directories” chapter on how paths are processed in TML-g.

Examples

Invoke an external template to create a file (the parameter *n*=5 is passed to the template):

```
\create{
    template="invert_matrix.tml"
```

```
    output="invert_matrix.asm"
    n=5
}
```

Invoke a snippet, place its output to the current output stream (since *output* parameter is not specified):

```
\create{
    snippet="namerecord"
    name="John"
    surname="Smith"
    birth=1970
    occupation="traffic regulation officer"
}
```

(You can see the definition of this snippet under the “Examples” section of \snippet tag description).

\copy

(content type: *arguments*)

Can be used to copy files. The following arguments must be specified:

- *source* (string) – source file name or path;
- *destination* (string) – destination file name or path.

The file name can include absolute or relative path. TML-g interpreter will create output directories automatically if needed. See “Paths and directories” chapter on how paths are processed in TML-g.

Examples

```
\copy{
    source="src/Makefile"
    destination="makefile"
}
```

\parameters

(content type: *subtags*)

Used to specify what input parameters should be given to the current template or snippet. It should be normally placed before any of these parameters is actually referenced. Using this tag is non-obligatory but strongly recommended because it possesses the following functionality:

- issue an error when a parameter marked as required wasn't supplied by the caller;
- issue an error when excess parameters (undeclared in the \parameters tag) are erroneously passed to the template or snippet;
- ensure that the parameter would have an expected type (converting between string and number types when appropriate);
- set a parameter to its default value when it is marked as optional and wasn't supplied by the

caller.

Also, looking at this tag is an easy way for developer to learn what parameters must be supplied to the template or snippet. It is therefore somehow similar to the function declaration in programming languages.

Parameters are specified by using nested tags `\req` (for required parameters) and `\opt` (for optional parameters). Any number of `\req` and `\opt` tags can be present.

\req and \opt

(content type: *arguments*)

These tags specify requred or optional parameter inside the `\parameters` tag. The following arguments should be set:

- *name* (string) – specifies parameter name;
- *type* (string) – optional, specifies parameter type (number, string, boolean or table);
- *default* (type may vary) – optional, valid only for `\opt` tag, specifies default value that is set automatically when the caller doesn't pass this parameter.

Examples

```
\parameters{
     \req{name="name";type="string"}
     \req{name="surname";type="string"}
     \req{name="birth";type="number"}
     \opt{name="occupation";type="string";
          default="unemployed"}
}
```

3.4. Branching and loop control tags

\if, \then, \else and \elsif

(content type: *string* for `\if` and `\elsif`, *freetext* for `\then` and `\else`)

These tags have quite an obvious meaning and are used to control branch processing. `\if` and `\elsif` tags evaluate their content as Lua expression. Any non-empty string, non-zero number or boolean *true* results in the nearest next `\then` branch being processed; any other values result in `\else` or `\elsif` branch being processed.

Although it is possible to insert other tags (irrelevant to branching) between `\if`, `\then`, `\elsif` and `\else`, such coding style is strongly discouraged.

Examples

```
\if{n<=0}
\then{\error{n must be at least 1!}}
\elsif{n>10}
```

```
\then{\error{n can't be greater than 10}}
\else{\echo{Starting processing...}}
```

\loop

(content type: *freetext*)

Organizes an infinite loop (which can be stopped by the \breakif tag when the user-defined condition is fulfilled). Nested loops are possible.

\breakif

(content type: *freetext*)

Evaluates its content as a Lua expression and immediately exits from the current cycle if the expression value is boolean *true*, non-empty string or non-zero number.

Examples

```
\format{strict=true}

\script{i=1}
\loop{
     Number \eval{i}\x{n}
     \breakif{i==5}
     \script{i=i+1}
}
```

The above code generates the following output:

```
Number 1
Number 2
Number 3
Number 4
Number 5
```

3.5. Other tags

\env

(content type: *string*)

Query an environment variable value.

Examples

```
\env{PATH}
```

\system

(content type: *freetext*)

Execute a shell command via Standard C Library system() function.

Examples

```
\system{make -f Makefile.gcc}
```

\getinfo

(content type: *string*)

Query interpreter-related information. The following strings can be passed to `\getinfo`:

- inputpath – input (template) file path
- line – current line number in the template
- column – current column number in the template
- software – processing software name
- version – processing software version
- time – current date and time (in Standard C Library ctime() format)
- platform – platform name (“windows” and “unix” are currently supported).

Examples

```
Generated by: \getinfo{software} \getinfo{version}
At: \getinfo{time}
Platform: \getinfo{platform}
Template file path: \getinfo{inputpath}
```

\handler

(content type: *arguments*)

An advanced feature that can be used to create user-defined tags. The following arguments should be set:

- *tag* (string) – tag name
- *function* (string) – Lua handler function name

Lua handler function must be defined earlier. It takes no arguments and must use *tml_parsecontent* or similar functions to obtain a tag content or *tml_discardcontent* in order to ignore it.

Examples

```
\script{
     function handle_p()
     write("<"..tml_parsecontent()..">\\n");
     end
}

\handler{tag=”p”;function=”handle_p”}
```

```
\p{This text will be surrounded with angle brackets}
```

\config

(content type: *string*)

Points to the TML-c configuration file (see *TML-c semantics*). Should normally be ignored by a TML-g interpreter.

3.6. TML-g Lua extensions

In addition to basic Lua functions which are well described in Lua reference, TML-g defines several specific functions which are referred to as "Lua extensions":

write(*string*)

Takes 1 argument and writes it to the output stream (i.e. the data being written are used as tag content to the parent tag, or written to the output document if there's no parent tag).

tml_parsecontent()

Suitable only within tag handler functions (see \handler tag description). Returns tag content as a string.

tml_forwardcontent()

Suitable only within tag handler functions (see \handler tag description). Puts tag content directly to the output stream.

tml_discardcontent()

Suitable only within tag handler functions (see \handler tag description). Discards tag content. These function should be called if the content isn't needed.

tml_parseparameters()

Suitable only within tag handler functions (see \handler tag description). Parses tag content as an *argument list* and returns a Lua table containing all arguments.

tml_getparserdata(*string*)

Queries interpreter information. The returning value depends upon an input string:

- *indent* – returns current indentation string
- *strict* – returns *true* if strict formatting is being used or *false* otherwise (see \format tag description)
- *tagstack* – returns a Lua table containing names of current tag, parent tag and so on, up to the top level.

tml_setparserdata(*string,value*)

Configures interpreter. *indent* or *strict* strings are supported, as defined above.

3.7. Paths and directories

Some tags require user to specify a file name. If a path is specified alongside with a file name, both `'/'` and `'\\'` can be used as directory delimiters (regardless of platform).

If a user specifies a relative path or no path at all, TML-g interpreter tries to build an absolute path relative to the directories listed below.

When these files are meant to be used as input (i.e. \include, \includetext tags, *template* argument of the \create tag, *source* argument of the \copy tag), TML-g tries to use the following directories as bases, in this particular order:

1. Directory containing the currently processed file (when processing standard input, current directory is used instead).
2. Base template directory, as specified in tmlgen.conf (*TemplateBaseDir* parameter).
3. Template directories, as specified in tmlgen.conf (*TemplateDir* parameters).
4. Current directory

When these files are meant to be used as output (i.e. *output* argument of the \create tag, *destination* argument of the \copy tag), the following rules apply:

1. If the output directory has been set for the current template using *outputdir* argument of the \create tag of the calling template, that directory is used as a base.
2. Otherwise if TML-g interpreter is currently writing output to the file, the directory containing that file is used as a base. If TML-g interpreter writes to the standard output, current directory is used as a base.
3. TML-g interpreter will create any directory contained in the output path automatically if it doesn't exist.

4. TML-c semantics

TML-c is a supplementary language flavour designed to describe user interfaces facilitating entry of TML-g template parameters. Note that TML-c is still in somewhat preliminary state.

The software such as TMLconf uses TML-c description to build a configuration GUI for a user. Based on user's choice, an *invocation script* is then created. Invocation script is a small TML-g template that contains a \config tag pointing to the original TML-c user interface description, and a \create tag which instantiates the appropriate TML-g template.

4.1. Basic TML-c tags

\title

Configuration window title.

\template

Path to the template (TML-g) file, can be absolute or relative. Both forward slash (`'/'`) and backslash (`'/'`) can be used in the path.

\refmode

Specifies whether the \config tag of the invocation script should contain absolute or relative path to the current TML-c file. String content can be *absolute* or *relative*, default value is *relative*.

\autosave

Specifies whether the invocation script should be saved. String content can have the following values:

- *true* – the invocation script is saved to the file with ".tmi" extension appended;
- *false* – the invocation script is saved to the temporary file that is deleted immediately after completion.

Default is *false*.

4.2. Control tags

\control

Defines a GUI control which allows user to set TML-g template parameter value. All the necessaty data are specified with nested tags.

\name

A nested tag for the \control parent. String content is a template parameter name associated with the control.

\type

A nested tag for the \control parent. String content is a type of the template parameter associated with the control, can be *string*, *number* or *boolean*.

\widget

A nested tag for the \control parent. String content is a widget type. The following widget types are currently supported:

- *text* – a simple text input field;
- *combo* – a combobox (drop-down list);
- *sfn* – save file name; to be used for *output* parameter or other template parameters specifying output file names;
- *ofn* – open file name; to be used for those template parameters that specify input file names;
- *hidden* – hidden widget, can't be modified by the user, always has default value.
- *directory* – allows the user to choose a directory, useful for the *outputdir* template argument.

\descr

A nested tag for the \control parent. String content is a text description of a control.

\option

A nested tag for the \control parent. Suitable only for a *combo* widget type. Its subtags define one combobox option:

- \value – option value, the parameter associated with the control will be assigned to this value if a user selects this option;
- \descr – option description.

One \control tag can have several \option subtags.

\default

(content type: *string*)

A nested tag for the \control parent. Specifies a parameter default value.

4.3. TML-c example

The following example configures three parameters for some *template.tml* template using three widgets: ofn, text and combobox.

```
\title{Configuration GUI}
\template{template.tml}
\control{
     \name{output}
     \type{string}
     \widget{sfn}
     \descr{Output file name}
}
\control{
     \name{string_parameter}
     \type{string}
     \widget{text}
     \descr{String parameter}
     \default{some text}
}
\control{
     \name{combo_parameter}
     \type{number}
     \widget{combo}
     \descr{Combo parameter}
     \option{\value{1}\descr{One}}
     \option{\value{2}\descr{Two}}
     \default{2}
}
```

An invocation script generated from the above description (assuming that user accepted default parameter values):

```
\config{gui.tmc}
\create{
     template="template.tml"
     string_parameter="some text"
     combo_parameter=2
}
```

Note that *gui.tmc* in the above example is a name of the TML-c template.