CoreTML framework 1.4

Software Manual

2010-2015

http://coretml.sourceforge.net

1. Introduction

CoreTML framework is an open-source template-based configuration system. It takes a template (or templates) along with user-supplied parameters and generates an output file (or files).

Although it can be used for many purposes, CoreTML framework was created primarily to support configurable semiconductor IP core creation in such languages as VHDL and Verilog.

CoreTML templates use the Template Markup Language (TML) which comes in two flavours: TML-g for templates themselves and TML-c to describe graphical interface that allows a user to configure template parameters. TML-g makes extensive use of Lua programming language (see http://www.lua.org).

Output files are usually generated in two stages.

1. An *invocation script* containing template parameters must be created. Invocation scripts can be generated by TMLconf based on user's choice.
2. TMLgen processes an invocation script and template file(s) and generates output file(s).

The basic CoreTML workflow from the end user viewpoint is shown on the below diagram. TMLconf and TMLgen are components of the CoreTML software package; it is the developer's responsibility to create a TML-c user interface description and TML-g template.

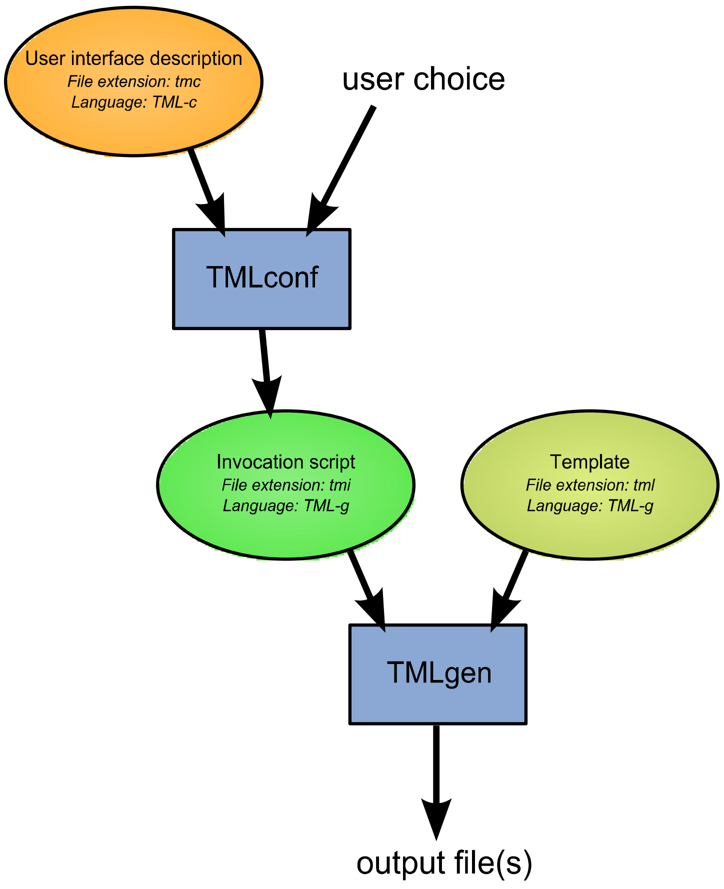


Fig. 1: Generic CoreTML workflow (from the end user viewpoint)

2. CoreTML components

CoreTML framework includes the following pieces of software:

- *TMLgen* – the main program used to process TML-g templates;
- Supplementary tools:
  - *maketml* – a simple tool that can be used to generate syntactically correct TML-g template from any text file;
  - *gentmc* – a simple tool that parses a TML-g template and creates an user interface description (TML-c) for TMLconf.
- *TMLconf* – a GUI program allowing the user to configure template parameters and run TMLgen with these parameters (invocation script is created automatically);
- *CoreTML editor* – a fork of SciTE (see http://www.scintilla.org/SciTE.html) featuring TML syntax highlighting and a few other TML-specific changes;

TMLgen and supplementary tools are platform-neutral and can be used both under Microsoft Windows and under major Unix-like systems. *TMLconf* and *CoreTML Editor* in current state serve mainly demonstration purposes and support only Microsoft Windows.

3. TMLgen

3.1. Overview

TMLgen is a command-line program that processes TML-g templates. It is a core part of the CoreTML framework.

3.2. Command line syntax

```
tmlgen [options] [inputfile] [outputfile]
```

If *outputfile* is omitted, TMLgen writes output data to the standard output stream. *inputfile* is a mandatory argument unless –`stdin` option is specified (see below).

Options:

- `--set <parameters>` – pass template parameters to TMLgen. `<parameters>` must be a string formatted as `par=value` (to pass just one parameter) or `par1=val1;par2=val2;...` (to pass a set of parameters). String parameter values must be surrounded by single quotes. If at least one parameter has string value, the whole `<parameters>` string must be surrounded by double quotes, e.g.

  ```
  tmlgen --set "title='Mr.';name='Smith'" example.tml
  ```

  Alternatively, parameter values can be surrounded by double quotes preceded by backslashes instead of single quotes.

  Note that there is a more powerful way to pass parameters than using `--set` option, namely via so-called *invocation scripts* (see below).
- `--noreq` – forcibly treat required template parameters as optional. This parameter is intended to assist template development. End users should exercise care when using this parameter since the template can behave unpredictably.
- `--lua` – invoke Lua interpreter instead of TML processor. *inputfile* must be a valid Lua program. TML-g Lua extensions can still be used in this mode.
- `--config <path>` – use the specified configuration file instead of default one.
- `--stdin` – use standard input instead of *inputfile*
- `--version` – display TMLgen version number end exit.
- `--luaversion` – display Lua interpreter version number end exit.
- `--help` – display a short help message and exit.

3.3. Configuration file

There is a configuration file that can be used to configure some TMLgen settings. By default (i.e. if the `--config` option hasn't been specified) TMLgen looks for *tmlgen.conf* in the following directories:

- Under Microsoft Windows, the *%APPDATA%\CoreTML\* is checked; if tmlgen.conf can't be found there, the *<INSTALL_PREFIX>\share\coretml* directory is used (*<INSTALL_PREFIX>* is a parent directory relative to the directory containing tmlgen executable).

- Under Unix-like systems, *~/.coretml/* is checked, if tmlgen.conf can't be found there, the *<INSTALL_PREFIX>/share/coretml/* directory is used (*<INSTALL_PREFIX>* is set during CoreTML build, default is */usr/local*).

The following variables can be set in the configuration file:

- *TemplateBaseDir* – base template directory. When TML-g template invokes another template and no absolute path for the invoked template is specified, TMLgen first assumes that the specified path is relative to the directory containing currently processed file. Failing that, TMLgen searches the template in the base template directory. If *TemplateBaseDir* is a relative path, it is build relative to the *<INSTALL_PREFIX>\share\coretml* directory (see above). There can be only one base template directory.
- *TemplateDir* – template directory. Similar to previous one, but defines an additional directory for template search. If *TemplateDir* is a relative path, it is assumed that *TemplateBaseDir* is a base. There can be any number of template directories.
- *EnforceNewlineStandard* – converts all the newline characters sent to the output stream. Can have the following values:
  - *windows* – use Microsoft Windows-like “\x0D\x0A” newlines;
  - *unix* – use Unix-like “\x0A” newlines;
  - *no* – don't do any conversion, leave newlines “as is”;
  - *platform* – use windows or unix newlines, depending on the platform TMLgen was compiled for. This is a default value.

  “*no*” value is not generally recommended since it can sometimes lead to a mixture of different newline endings.
- *DontSkipShebang* – by default, TMLgen ignores the first line in a TML template if it starts with `#!` (the so-called “shebang”). This feature can be used to instruct Unix-like systems to call TMLgen when executing the template. Set this variable to *true* to turn off this behavior. List of the possible values:
  - *true* – turn off special processing of shebang at the start of the file;
  - *false* – ignore shebang at the start of the file. This is a default value.

  Note 1: Shebang line is ignored only when reading a template from a file, but not when reading it from the standard input.

  Note 2: If you need to insert a shebang at the start of the output stream, you can use the `\x{g}` tag. See also the *Template Markup Language Reference*.
- Note 3: When executing a pure Lua script (with the `--lua` command-line argument), a shebang at the first line is always skipped, regardless of this option (this emulates standard lua interpreter behavior).

3.4. Using invocation scripts to pass template parameters

Any TML-g template can invoke another TML-g template using the \create tag. An invocation script is simply a TML-g template containing only one \create tag and producing no other output.

\create tag uses *parameter=value* pairs separated by newlines, semicolons, or both. There are two predefined \create tag parameters: *template* specifies template file path, *output* specifies output file path. If *output* is omitted, the template output is placed to the current output stream (defined by TMLgen command line).

Example:

```
\create{
    template="template.tml"
    output="out.txt"
    name="String parameter"
    n=7
}
```

\create tag syntax is covered in details in the *Template Markup Language Reference*.

4. Supplementary tools

4.1. maketml

maketml is the opposite of TMLgen. It takes a text file and produces a syntactically correct unparametrized TML-g template. Being processed by TMLgen, this template will generate exactly the same text file (with one exception: newline character formats can still differ if TMLgen is set co convert them). Templates generated by *maketml* can be used as a base to create a useful, parametrized TML-g template.

maketml ensures that all special characters such as backslashes and braces are properly escaped.

maketml command-line syntax:

```
maketml [options] inputfile [outputfile]
```

Options:

- `--overwrite` – overwrite output file;
- `--strict` – if this option is specified, the generated TML-g template will use strict formatting mode (see \format tag description in the *Template Markup Language Reference*).

4.2. gentmc

gentmc generates a basic TML-c user interface description to be used with TMLconf from a TML-g template. An input TML-g template must contain proper \parameters tag that describes template input parameters (see *Template Markup Language Reference*).

gentmc command-line syntax:

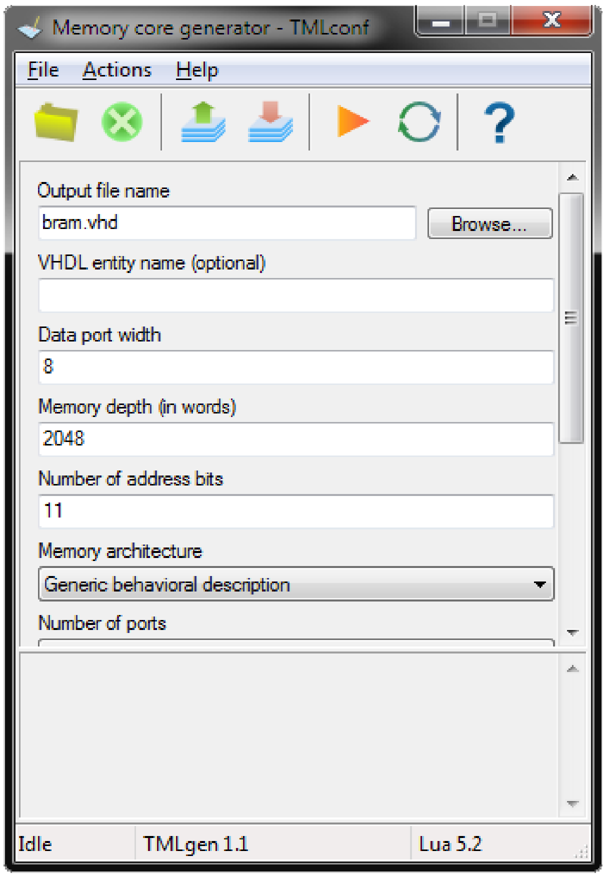
```
gentmc [options] inputfile [outputfile]
```

Options:

- `--overwrite` – overwrite output file;
- `--relative` – try to reference the template file using relative path (if possible).

5. TMLconf

TMLconf is a simple Windows-only application that uses user interface template contained in a TML-c file to create a GUI that allows the user to configure parameters for a TML-g template.



Output file name: bram.vhd
VHDL entity name (optional):
Data port width: 8
Memory depth (in words): 2048
Number of address bits: 11
Memory architecture: Generic behavioral description
Number of ports

Fig. 2: An example user interface created by TMLconf

TMLconf commands and features:

- Open/close a TML-c user interface template;
- Generate an invocation script that can be used to invoke TMLgen;
- Load an invocation script allowing the user to change parameters;
- Invoke TMLgen directly.

TML-c user interface description can be generated with *gentmc* (see above) or created manually. More information on TML-c is available in the *Template Markup Language Reference*.

6. CoreTML Editor

CoreTML Editor is basically a slightly modified version of the SciTE editor (see http://www.scintilla.org/SciTE.html). These modifications include:

- TML syntax highlighting;
- ability to remember most editor settings after exit;
- integration with the other CoreTML tools.

Most other features are borrowed from SciTE and will not be covered in this manual.

6.1. Creating a template

CoreTML Editor supports TML-g syntax highlighting even for unnamed files. It is possible to start writing TML-g template code and then save it as a file with the *.tml* extension.

If a template doesn't require any parameter, it is possible to invoke TMLgen for it simply by selecting *Run→Go* menu command or pressing the appropriate toolbar button. The TMLgen output will appear in the output window.

6.2. *Parametrized templates and* TMLconf

To run a parametrized template from the CoreTML Editor user interface, one needs first to create a TML-c description that will be used by TMLconf to ask user for template parameters. TML-c description can be created manually (see *Template Markup Language Reference*) or using *Run→Generate configuration interface description* menu command (available when a *.tml* file is opened). This command invokes *gentmc* to generate TML-c description; it works only if the source *.tml* template has a proper \parameters tag.

Invoking *Run→Go* for the TML-c description results in TMLconf windows being opened. Output files generated by TMLconf will be automatically opened in the CoreTML Editor.

6.3. *Invoking* maketml

It is possible to invoke *maketml* directly from the CoreTML Editor user interface. Use *Run→Generate TML template* or *Run→Generate TML template (strict formatting)* menu commands. *maketml* will create a syntactically correct but unparametrized TML template that will need some work to become useful.

6.4. Other editors

CoreTML Editor serves mainly demonstration purposes and it is unlikely that it will be developed any further. If you need an up-to-date, actively maintained editor, you may want to try AkelPad (http://akelpad.sourceforge.net) which is an open-source text editor extendable via a powerful plugin system. CoreTML Windows package contains AkelPad syntax highlighting definitions for TML languages as well as VHDL and Verilog (they are in the *share/extras/akelpad* directory). They must be copied to the appropriate AkelPad directory (AkelFiles\Plugs\Coder).

7. Building from source

7.1. Overview

Since version 1.2 CoreTML framework uses CMake (http://www.cmake.org/) as a build system. CMake is an open-source program that can generate build scripts for different toolchains (e.g. the GNU toolchain or Microsoft toolchain) as well as project directories for several popular IDEs (Microsoft Visual Studio, Eclipse CDT, Code::Blocks or KDevelop). Full list of supported generators is available in the CMake documentation.

CMake encourages out-of-tree builds, i.e. you should run CMake in a new directory and not in the source directory.

7.2. Prerequisites

In order to build CoreTML framework from source you should have:

1. CMake 3.4.0 or more recent version (can be downloaded from http://www.cmake.org).
2. One of the toolchains supported by CMake, for example:
   - For Unix-like systems: GNU toolchain
   - For Microsoft Windows:
     - Microsoft Visual Studio (inluding free Express versions)
     - MinGW
3. To build installer for Windows, you will also need NSIS (Nullsoft Scriptable Install System). As of early 2014 mainline NSIS builds can't handle string larger than 1024 bytes which can lead to problems when updating the PATH variable. Until they fix this limitation, using “Large strings” special build (http://nsis.sourceforge.net/Special_Builds) is strongly recommended.
4. Test system is currently dependent on GNU make. To be able to run regression tests under Microsoft Windows you need to have MSYS or Cygwin. Running tests is optional but strongly encouraged if you make modifications to the software.

7.3. Build options

Build options can be set using cmake command-line switches or with the help of the CMake GUI. The command-line syntax is:

```
cmake -D option_name=option_value ...
```

List of useful options:

- CMAKE_INSTALL_PREFIX – path that will be used to install CoreTML. Under Unix-like systems default prefix is */usr/local*; set this variable to install it elsewhere. Under Microsoft Windows you can usually ignore this variable.
- optionForceInternalLua (possible values: ON, OFF; default is ON) – TMLgen uses Lua to provide core functionality. By default, bundled version is used. Set to OFF to use the system-installed version (if present).
- optionNoTMLconf (possible values: ON, OFF; default is OFF) – don't build TMLconf.

Ignored under non-Windows systems.

- optionNoEditor (possible values: ON, OFF; default is OFF) – don't build the CoreTML Editor. Ignored under non-Windows systems.

7.4. Steps to build

1. Create build directory (it can be located anywhere, but we will assume that it is created in the top-level directory of the extracted source tree, alongside “src” directory):

   ```
   mkdir build
   ```

2. Enter build directory:

   ```
   cd build
   ```

3. Invoke Cmake:

   ```
   cmake ../src
   ```

   You can specify generator type here. For example, to create Microsoft Visual Studio 12 (2013) solution you can invoke:

   ```
   cmake -G "Visual Studio 12" ../src
   ```

4. For makefile generators, run “make”. Otherwise build the solution using the IDE of your choice.
5. To install, invoke “make install” for makefile generators or run INSTALL task for IDE generators.
6. To create an installer package, invoke “cpack”.

7.5. Notes

The following CMake generators have been tested at least once. Other generators should probably also work, but it is not guaranteed.

- *Unix Makefiles*

  This is pretty straightforward. Don't forget to set CMAKE_INSTALL_PREFIX option if you plan to install CoreTML framework to a location other than default (which is */usr/local*). For example, to install it to /home/user/coretml/:

  ```
  cmake -D CMAKE_INSTALL_PREFIX=/home/user/coretml ../src
  make
  make install
  ```

  Under Unix-like systems no other options or switches should be required.

  After that you can create DEB and/or RPM packages:

  ```
  cpack -G DEB
  cpack -G RPM
  ```

- *MSYS Makefiles*

  Both MSys and MinGW must be in the PATH.

  ```
  cmake -G "MSYS Makefiles" ../src
  make
  make install
  ```

- *MinGW Makefiles*

  MinGW must be in the PATH, whereas MSys must NOT be in the PATH, otherwise CMake will complain. Use mingw32-make instead of make.

  ```
  cmake -G "MinGW Makefiles" ../src
  mingw32-make
  mingw32-make install
  ```

- *NMake Makefiles*

  Environment variables for Visual C++ command-line tools must be set before running CMake (this can be accomplished with the *vcvars32.bat* script).

  ```
  vcvars32
  cmake -G "NMake Makefiles" ../src
  nmake
  nmake install
  ```

- *Visual Studio 9 2008*

  Environment variables for Visual C++ command-line tools must be set before running CMake (this can be accomplished with the *vcvars32.bat* script). If Visual Studio fails to build TMLconf or CoreTML Editor, try to turn off incremental linker in the project options.

  ```
  vcvars32
  cmake -G "Visual Studio 9 2008" ../src
  ```

- *Visual Studio 12*

  Environment variables for Visual C++ command-line tools must be set before running CMake (this can be accomplished with the *vcvars32.bat* script).

  ```
  vcvars32
  cmake -G "Visual Studio 12" ../src
  ```

7.6. Installation

Under Microsoft Windows the recommended way to install the CoreTML framework is to use a provided installer. To uninstall, just use the “Add/Remove Programs” tool in the Control Panel.

Under Unix-like systems, build the CoreTML framework from source and invoke “make install” as root. To uninstall, run “uninstall.sh” as root. You can also build DEB or RPM packages using CPack.

8. Using third-party Lua libraries

Lua is a minimalistic language, but it can be extended with the third-party libraries. There are quite a few of them at Lua-users (http://lua-users.org/wiki/LibrariesAndBindings) and the Kepler Project (http://www.keplerproject.org/). This chapter outlines using these libraries with TMLgen.

8.1. Using system Lua

Under Unix-like systems the recommended way is to install Lua 5.2 or later using package manager and link TMLgen against it. Then you can use standard management system (such as LuaRocks) to install the needed libraries.

8.2. Linking against bundled Lua

In order to use third-party libraries the bundled Lua interpreter must be built as a shared library (which is the default mode), otherwise Lua will fail providing “multiple VMs detected” error message. Lua headers will be installed to the *<INSTALL_PREFIX>/include/coretml* directory, libraries – to the *<INSTALL_PREFIX>/lib/coretml* directory (default install prefix is */usr/local*). Third-party library must be build as a shared object (DLL under Microsoft Windows) and linked to the provided import library.

Under Microsoft Windows, the same toolchain should be used to build both TMLgen and the third-party libraries.

For example, to build LuaFileSystem (http://keplerproject.github.io/luafilesystem/) using MinGW one can invoke:

```
>git clone https://github.com/keplerproject/luafilesystem.git lfs
>cd lfs\src
>gcc "-Ic:\Program Files (x86)\CoreTML\include\coretml" -shared
lfs.c "c:\Program Files (x86)\CoreTML\lib\coretml\liblua.dll.a" -o
lfs.dll
```

To build the same library with the Visual C++ (assuming that TMLgen was also built using Visual C++):

```
>vcvars32
>git clone https://github.com/keplerproject/luafilesystem.git lfs
>cd lfs\src
>cl "/Ic:\Program Files (x86)\CoreTML\include\coretml" /c lfs.c
>link /DLL /DEF:lfs.def lfs.obj "c:\Program Files
(x86)\CoreTML\lib\coretml\lua.lib"
```

To use the built DLL, copy it to the CoreTML *bin* directory.

9. Software version table

CoreTML framework 1.3 includes the following software versions:

Software	Version included
TMLgen	1.4
with Lua	5.3.1
maketml	1.0
gentmc	1.0
TMLconf	1.4
CoreTML Editor	1.0.1
based on SciTE	2.11