WUPPER Code Generator Documentation

Release 0.6.2

Mark Dönszelmann, Jose Valenciano, Jörn Schumacher

Oct 22, 2017

CONTENTS

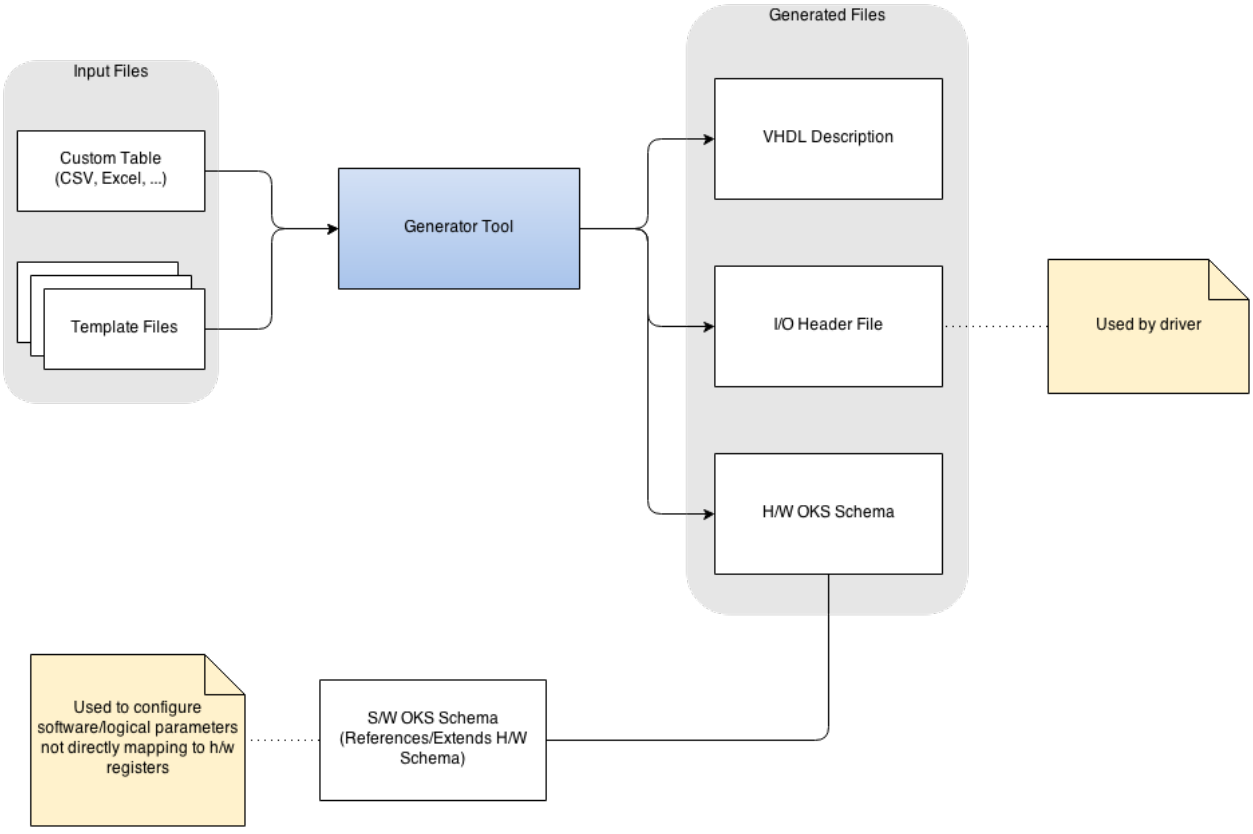
The Wupper Code Generator is a tool to generate various files used for the implementation of configuration mechanism of the FELIX interface card firmware. Files that can be generated include:

- A VHDL description of configuration parameters with required datatypes (e.g., a list of VHDL record types).
- A VHDL file for a register mapping, which relates parameter register to PCIe addresses. This can include write- and read-processes for all configuration registers.
- A matching C header file, which matches the register mapping described in the previous point to C datastructures which represent the different configuration options.
- A (partial) OKS schema for the firmware parameters. This still needs to be extended with an OKS schema for the software configuration options. It is yet to be determined how much of the H/W OKS schema can be autogenerated, as not all parameters of the firmware should be configurable using an OKS database.
- Other use cases are thinkable, e.g., automatic documentation of the parameters, registers, datatypes, etc.

The Wupper Code Generator tool uses two kinds of inputs:

- A configuration file describing all configuration registers of the firmware, as well their data types, default values, and other relevant information. The format of the input file is YAML.
- A (set of) template file(s), describing how the output of the tool should look like. Jinja2 is used as template engine.

Additionally, the tool automatically generates a unique address for each register. This can be used to assign individual registers to addresses in the PCI memory mapping.



CHAPTER

ONE

CONTENTS

1.1 Installation

WupperCodeGen is written in Python and was tested with versions 2.6 and 2.7. Additionally, you will need to install the Jinja2 template engine and the Python Yaml parser. In case of Python 2.6 you will have to install the argparse module as well.

1.1.1 Installation from PyPI using pip

Installation using pip, which will install all dependencies:

```
pip install wuppercodegen
```

1.1.2 Scientific Linux and RedHat-based distributions

Installation of system-wide Jinja2 in SLC6:

```
sudo yum install python-jinja2 python-argparse python-yaml python-markupsafe
```

1.1.3 Ubuntu

Installation of system-wide Jinja2 in Ubuntu:

```
sudo apt-get install python-jinja2 python-yaml python-markupsafe
```

1.1.4 Using Virtual Python Environments

This method allows to install the necessary dependencies in an isolated Python environment, without affecting system packages. The method described here is independent of the operating system being used, as long as it is used with a supported version of Python.

Installation in a virtual environment:

```
# create the virtual environment
virtualenv venv/ --python=/path/to/python26-or-27

# activate the virtual environment
source venv/bin/activate
```

```
# install Jinja2 ad other modules
      pip install Jinja2
pip install argparse
pip install PyYAML
pip install MarkupSafe

# you can now use WupperCodeGen
# the virtual environment can be deactivated with
source venv/bin/deactivate
```

1.2 User Documentation

1.2.1 Command Line Interface

A Makefile is provided that calls WupperCodeGen with the correct parameters to generate the output files. A simple `make` in the WupperCodeGen directory is enough to produce the output, which is stored in the directory 'output'.

If there is a need to do so, the tool can be called manually in the following way:

```
usage: wuppercodegen [-h] [--version] config_file template_file output_file
```

The following parameters have to be set:

`config_file` An input file containing a description of all configuration registers and grouping thereof in YAML format.

`template_file` A Jinja2 template file, used to describe the output file.

`output_file` The filename of the output file that is to be generated.

1.2.2 Input Files

Input files are located in the current directory. There are two types of input files:

- data files, containing a description of the desired bitfields, registers and groups in YAML format,
- template files, describing how the generated code should look like.

Users normally only deal with the YAML file.

The Register Description File is a YAML ('registers.yaml') file containing a description of Groups, Registers and Bitfields. The top-level group is named 'Registers'. Groups are named, and may contain other Groups and or Registers. Groups can be repeated to create a Sequence of Groups/Registers. Registers are also named and contain one or more Bitfields. Bitfields are named if there are more than one. Groups, Registers and BitFields are commonly referred to as Nodes. Below a description of each of these types:

Group

Groups are declared in yaml by their name, followed by a number of key-value pairs, referred to as attributes. The only obliged attribute is 'entries' which lists all the subgroups or registers contained in this group. 'entries' is a list. The top-level group is called 'Registers', all other groups can have any name, but it needs to be unique.

Subgroups are declared just like groups and referred to by name in the 'entries' array under the key 'ref'. A group that needs repeating may contain the key 'number', which is added as a index-parameter in the lookup of the subgroup, and used in the name of its registers, see under registers. Below is group example:

```
Registers:
  type: R
  step: 0x010
  default: 0
  entries:
    - ref: Bar0
      offset: 0x0000
    - ref: Bar1
      offset: 0x0000
    - ref: Bar2
      offset: 0x0000

Bar0:
  entries:
...
```

Register and Bitfields

Registers are declared directly inside the 'entries' attribute of a group. They need a 'name' and a 'bitfield' attribute. Bitfield is in fact a list of one or more bitfields, which must contain 'range' as attribute and 'name', if there is more than one bitfield in the list. Range is given as a single bit or as a range of high to low bits (both included in the range).

Below is a register and bitfield example:

```
GenericBoardInformation:
  entries:
    - name: BOARD_ID
      bitfield:
        - range: 79..64
          name: SVN
          desc: Build SVN Revision
        - range: 39..0
          name: BUILD
          desc: Build Date / Time in BCD format YYMMDDhhmm

    - name: STATUS_LEDS
      type: W
      bitfield:
        - range: 7..0
          default: 0xAB
          desc: Board GPIO Leds

...
```

The name attribute of a register may contain one % operator to use the repeat-index-parameter to give repeated registers a unique name.

```
Bar1:
  entries:
    - ref: INT_VEC
      number: 8
...
```

```
INT_VEC:
  entries:
    - name: INT_VEC_%01d
...
```

Attributes

Attributes in Bitfields, Registers and Groups are inherited, which means that if you set them on a group, all sub-groups, sequences, registers and bitfield inherit them. This way you can set defaults for a particular group. The following attributes do NOT inherit: 'name', 'full_name', 'offset', 'address', 'index', 'entries', 'number', 'ref', 'is_bitfield', 'is_register', 'is_group'.

The following attributes are used/available:

name (not inherited) The name of the bitfield, group or register. The name may include fields such as {index} for sequences and {bitfield} for bitfield names.

full_name (not inherited)

- For Sequences, the formatted name of the sequence name and the index, if the sequence name includes {index}.
- For Registers, the formatted name of the register name and the index, if the register name includes {index}.
- For Bitfields, the formatted name of the parent register and the bitfield name if the register name includes {bitfield}. The bitfield part is separated from the name by an underscore. Bitfields of which the parent register is part of a sequence may use also {index} in the parent register's name to format that name.

dot_name (not inherited)

- For Bitfields only, separating the bitfield part from the name by a dot.

prefix_name (not inherited)

- For Registers only, the part of the name without the {bitfield} specifier. To be used if you just want the generic name of the register.

desc A description of the register or bitfield. The '|' operator can be used for multi-line descriptions, which if used as end-of-line comments in the template will properly format.

type The type of register/bitfield, which can be R: read-only, W: read/write or T: trigger (any write results in something happening, e.g. reset...)

default The default value to be used to initialize the register or bitfield. Default can be given as an array, if needed. All fields should be supplied, for example a 6 entry array [0..5]:

```
default:
  - 0x00000124aaaa8006
  - 0x00000124aaaa8078
  - 0x00000124aaaa8001
  - 0x00000124aaaaff80
  - 0x00000124aaaae628
  - 0x00000124aaaa99d0
```

value Value (or string) to be used for Trigger registers.

offset (not inherited) Sets the offset used for a register or group.

address (not inherited, calculated) Address is a calculated value. It starts at 0 for the top-level group. The 'offset' attribute gets added to the address of a group to create the current address. Sub-groups and registers are located

at the current address. The current address is incremented by 'step' for every register. One can place any register or group at a particular address by calculating the needed offset from the address of the parent group.

step The address increase added to offset for every register.

entries (not inherited) A list of registers or references to sub-groups.

ref (not inherited) A reference (by name) to a sub-group.

number (not inherited) Number of times a referred group needs to be repeated in a sequence.

index (not inherited) The index for this group or register inside a sequence.

bitfield List of (at least one) bitfields describing the bits in a register.

range The bit range (even single bit) of a bitfield. Ranges can be expressed as single integers, as a high..low range, or with the word 'any'.

any other The value given, inherited from group to group to register to bitfield.

Metadata

You can use the following metadata which is available in the Metadata dictionary:

version The version of WupperCodeGen

name: The name of WupperCodeGen

exec: The program executed

config: The path of the config file

template: The path of the template file

output: The path of the output file

cmdline: The full command line

Each of these can used inside the config file or in the template. Care must be taken when the output is in LaTeX as the codes change for LaTeX, so any variable in the config file (the template file is already coded for LaTeX) must be filtered through the tex_yaml_encode filter.

Below an example of setting up a warning how the output file is generated:

```
Registers:
  version: 1.3.0
  warning: |
    This file was generated from '{{ metadata.template }}', version {{ tree.version }}
    by the script '{{ metadata.name }}', version: {{ metadata.version }}, using the␣
↪following commandline:

    {{ metadata.cmdline }}

    Please do NOT edit this file, but edit the source file at '{{ metadata.template }}
↪'

  entries:
    ...
```

and the usage in a LaTeX the template

```
((( tree.warning|tex_yaml_encode|tex_comment )))
...
```

and its output

```
% This file was generated from 'warning.tex.template', version 1.3.0
% by the script 'WupperCodeGen', version: 0.3, using the following commandline:
%
% ../wupper_codegen.py warning.yaml warning.tex.template warning.tex
%
% Please do NOT edit this file, but edit the source file at 'warning.txt.template'
...
```

1.3 Developer Documentation

Developers manage the code generated by WupperCodeGen. The output is defined by Jinja2 template files, usually ending in '.template'.

1.3.1 Template Files

The template files should be written using Jinja2 syntax. Jinja2 statements can be either flow control commands such as `{% for g in groups %}` or simple text substitutions such as `{{ g.name }}`. For a detailed description of the Jinja2 language, please refer to the official documentation, but look below for the special codes to use for LaTeX templates.

The data retrieved from the input file (yaml) is available as variables in the template. Those variables are listed in the user documentation. Apart from these variables one may call a number of functions and filters as explained below.

The following global variables are available:

metadata A dictionary with some metadata, available for the config file and the template.

tree The root of all the nodes, e.g. the one named 'Registers'.

registers A list of all the registers. All registers are linked into the tree by their 'parent' attribute.

nodes A lookup table of all nodes, stored by name or full_name. All nodes are linked into the tree by their 'parent' attribute.

In case wuppercodegen is called in "diff" mode you also have access to the following global variables:

diff_tree The root of all the nodes from the diff file, e.g. the one named 'Registers'.

diff_registers A list of all the registers from the diff file. All registers are linked into the diff_tree by their 'parent' attribute.

diff_nodes A lookup table of all nodes from the diff file, stored by name or full_name. All nodes are linked into the diff_tree by their 'parent' attribute.

changed_registers A list of changed registers.

1.3.2 Functions

All functions of Node can be called on BitField, Register, Group or Sequence. These themselves have extra functions available as well. Functions without arguments can be called as if they were attributes (no parentheses). Classes.

class `wuppercodegen.classes.`**Node** (*parent*, *dictionary*, *name*)

Bases: `object`

Root object of BitField, Register, Group and Sequence.

full_name()
Return 'print_name' if 'format_name' is defined, otherwise it returns 'name'.

has_endpoint(*endpoint*)
Return if endpoint is supported.

has_groupname()
Return True if group (not sequence) and 'group' is defined for a groupname.

has_read_bitfields()
Return True if a register has any read bitfields.

has_trigger_bitfields()
Return True if a register has any trigger bitfields.

has_write_bitfields()
Return True if a register has any write bitfields.

is_bitfield = False
True is node is a bitfield.

is_group = False
True if node is a group.

is_read()
Return True if a group, register or bitfield is readable.

is_register = False
True if node is a register.

is_sequence = False
True if node is a sequence. Sequences are also groups.

is_trigger()
Return True if a group, register or bitfield is triggerable.

is_write()
Return True if a group, register or bitfield is writeable.

print_name()
Return formatted name according to 'format_name', using 'index', 'name' and 'parent.print_name'.

class wuppercodegen.classes.**BitField**(*parent*, *dictionary*, *name*)
Bases: *wuppercodegen.classes.Node*

Bitfield, defines specifics about bits inside the register.

bits()
The number of bits in the bitfield range. Returns 0 if the range is 'any'.

dot_name()
Return register name with bitfield name appended with a '.'.

full_name()
Return 'print_name' if 'format_name' is defined, otherwise appends bitfield 'name' with underscore to 'name'.

hi()
The highest bit in the bitfield range. Returns 0 if the range is 'any'.

lo()
The lowest bit in the bitfield range. Returns -1 if the range is 'any'.

print_name()
Return formatted name according to 'format_name', using 'index', 'name' and 'parent.print_name'.

value_string()
Return value indexed if needed.

class wuppercodegen.classes.**Entry**(*parent*, *dictionary*, *name*, *index*)
Bases: *wuppercodegen.classes.Node*

Allows to be an entry in a sequence.

index = None
Index if part of a sequence

size()
Return the size in bytes of the Register, Group or Sequence.

class wuppercodegen.classes.**Register**(*parent*, *dictionary*, *name*, *index*)
Bases: *wuppercodegen.classes.Entry*

Register.

bits()
The number of bits in the bitfield range. Returns 0 if the range is 'any'.

full_name()
Return 'print_name' if 'format_name' is defined, otherwise returns 'name' formatted with 'index' if defined.

hi()
Return the highest of all bitfield ranges.

lo()
Return the lowest of all bitfield ranges.

prefix_name()
Return 'full_name' without the {bitfield} part.

size()
TBD.

sort_by_address()
TBD.

class wuppercodegen.classes.**Group**(*parent*, *dictionary*, *name*, *index*)
Bases: *wuppercodegen.classes.Entry*

Groups a number of registers.

size()
TBD.

class wuppercodegen.classes.**Sequence**(*parent*, *dictionary*, *name*, *number*, *index*)
Bases: *wuppercodegen.classes.Group*

Array of registers.

full_name()
Return name, formatted by index if defined.

number = None
Size of the sequence

1.3.3 Tests

Tests can be called if output is based on some condition. Tests.

wuppercodegen.test.**in_group**(*node*, *name*)
Return True if this bitfield, register, group or sequence belongs to 'group'.

1.3.4 Filters

Filters are used to modify input (with or without parameters). They are handy for formatting and aligning the output. Filters.

wuppercodegen.filter.**append**(*value*, *postfix*)
Format the input value as *value*postfix.

wuppercodegen.filter.**c_comment**(*value*, *indent=0*)
Split the input value in separate lines and indents each of them by 'indent' spaces.

Every line is surrounded by a c comment delimiter (/* ... */).

wuppercodegen.filter.**c_hex**(*value*, *digits=4*)
Format the input value as hexadecimal: 0x1F40.

wuppercodegen.filter.**c_mask**(*bitfield*)
Return the mask value based on the bitfield.hi and bitfield.lo values.

wuppercodegen.filter.**c_string**(*value*)
Escape YAML (multi-line) string into a C string.

wuppercodegen.filter.**camel_case_to_space**(*name*)
Convert CamelCase to space separated text.

wuppercodegen.filter.**cpp_comment**(*value*, *indent=0*)
Split the input value in separate lines and indents each of them by 'indent' spaces.

Every line is prepended by a comment delimiter (//).

wuppercodegen.filter.**dec**(*value*, *dec=1*)
Decrement value by 'dec'.

wuppercodegen.filter.**hex**(*value*, *digits=4*)
Format the input value using 'digits' in hexadecimal.

wuppercodegen.filter.**html_comment**(*value*, *indent=0*)
Split the input value in separate lines and indents each of them by 'indent' spaces.

The whole section is prefixed and suffixed by comment delimiters (<!– and –>).

wuppercodegen.filter.**html_string**(*value*)
Escape YAML (multi-line) string into html.

wuppercodegen.filter.**inc**(*value*, *inc=1*)
Increment value by 'inc'.

wuppercodegen.filter.**line_comment**(*value*, *prefix*, *indent=0*, *suffix=''*)
Generate line comment.

wuppercodegen.filter.**list_nodes_and_sequences**(*node*, *list=None*)
List the input group recursively.

A sequence is an end node. Children of groups agreed listed, but children of sequences are not. This filter can be used to generate a top-level list of registeres, referring to the output of list_sequences.

wuppercodegen.filter.**list_nodes_recursively**(*node*, *doc=False*, *list=None*)
List the input group recursively.

Groups are listed before their children. Bitfield are NOT listed. If 'doc' is true, then registers with the 'nodoc' attribute (sequences) are not in the list, but an artificial group with name (...) is inserted where registers are left out. The latter is used for documentation.

wuppercodegen.filter.**list_sequences**(*node*)
List the input group recursively.

Sequences and groups are listed before their children. Sequences are unwrapped, but only listed once. Bitfield are NOT listed. This filter can be used to generate structs in C/C++.

wuppercodegen.filter.**multi_line_comment**(*value*, *prefix*, *postfix*, *indent=0*)
Generate multi-line comment.

wuppercodegen.filter.**prepend**(*value*, *prefix*)
Format the input value as prefix*value*.

wuppercodegen.filter.**semi**(*field*, *semi=True*)
Append a semicolon unless append = False.

wuppercodegen.filter.**tex_comment**(*value*, *indent=0*)
Split the input value in separate lines and indents each of them by 'indent' spaces.

Every line is prepended by a comment delimiter (%).

wuppercodegen.filter.**tex_escape**(*value*)
Escape the input value for LaTeX.

wuppercodegen.filter.**tex_string**(*value*)
Escape YAML (multi-line) string into LaTeX and calls tex_escape.

wuppercodegen.filter.**tex_yaml_encode**(*value*)
Encode the standard codes (see below) as codes used in a LaTeX template.

Use this filter for values of attributes set in the config file and used in a LaTeX template.

wuppercodegen.filter.**version**(*value*)
Convert MajorVersion.MinorVersion to MajorVersion*0x100+MinorVersion.

wuppercodegen.filter.**vhdl_comment**(*value*, *indent=0*)
Split the input value in separate lines and indents each of them by 'indent' spaces.

Every line is prepended by a comment delimiter (–).

wuppercodegen.filter.**vhdl_constant**(*value*, *bits=1*)
Format the input value using 'bits' in binary of hexadecimal for VHDL.

wuppercodegen.filter.**vhdl_downto**(*bitfield*)
Format the input value as (*hi* downto *lo*).

wuppercodegen.filter.**vhdl_logic_vector**(*bitfield*)
Format the bitfield value as std_logic_vector(*hi* downto *lo*).

wuppercodegen.filter.**vhdl_value**(*bitfield*, *prefix*)
Return vhdl value.

If the input value is a trigger, the input.value is returned (and must be specified in the YAML file) either as String or as constant and will be vhdl formatted. If the input value is not a trigger, then vhdl_downto is called prepended by prefix.

wuppercodegen.filter.**xhex**(*value*, *digits=4*)
Format the input value as hexadecimal: 0x1F40. Specially for usage in latex where underscores do not work.

1.3.5 Codes for LaTeX (templates where the output ends in .tex)

As LaTeX uses a lot of special characters WupperCodeGen redefines the standard JinJa2 delimeters to some others. Care must also be taken in a LaTeX template to escape all texts. An escape_tex filter is available to handle this.

Delimiters	Standard	LaTeX
Statements	{% ... %}	((* ... *))
Expressions	{{ ... }}	(((...)))
Comments	{# ... #}	((= ... =))
Line Statements	# ... ##	

PYTHON MODULE INDEX

W

INDEX

R

S

T

V

W

X