cogeno Documentation

Release 0.2.1

Bobby Noelte

Mar 01, 2020

CONTENTS:

For some repetitive or parameterized coding tasks, it's convenient to use a code generating tool to build code fragments, instead of writing (or editing) that source code by hand.

Cogeno, the inline code generation tool, processes Python 3 or Jinja2 script "snippets" inlined in your source files. It can also access CMake build parameters and device tree information to generate source code automatically tailored and tuned to a specific project configuration.

Cogeno can be used, for example, to generate source code that creates and fills data structures, adapts programming logic, creates configuration-specific code fragments, and more.

CHAPTER

ONE

ABOUT

1.1 Introduction

1.1.1 About cogeno

Script snippets that are inlined in a source file are used as code generators. The tool to scan the source file for the script snippets and process them is cogeno. Cogeno and part of this documentation is based on Cog from Ned Batchelder.

The inlined script snippets can contain any Python 3 or Jinja2 code, they are regular scripts.

All Python snippets in a source file and all Python snippets of included template files are treated as a python script with a common set of global Python variables. Global data created in one snippet can be used in another snippet that is processed later on. This feature could be used, for example, to customize included template files.

Jinja2 snippets provide a - compared to Python - simplified script language.

An inlined script snippet can always access the cogeno module. The cogeno module encapsulates and provides all the functions to retrieve information (options, device tree properties, CMake variables, config properties) and to output the generated code.

Cogeno transforms files in a very simple way: it finds snippets of script code embedded in them, executes the script code, and places its output combined with the original file into the generated file. The original file can contain whatever text you like around the script code. It will usually be source code.

For example, if you run this file through cogeno:

```
/* This is my C file. */
...
/**
 * @code{.cogeno.py}
 * fnames = ['DoSomething', 'DoAnotherThing', 'DoLastThing']
 * for fn in fnames:
 *     cogeno.outl(f'void {fn}();')
 * @endcode{.cogeno.py}
 */
/** @code{.cogeno.ins}@endcode */
...
```

it will come out like this:

```
/* This is my C file. */
...
/**
 * @code{.cogeno.py}
 * fnames = ['DoSomething', 'DoAnotherThing', 'DoLastThing']
```

(continues on next page)

(continued from previous page)

```
 * for fn in fnames:
 *     cogeno.outl(f'void {fn}();')
 * @endcode{.cogeno.py}
 */
void DoSomething();
void DoAnotherThing();
void DoLastThing();
/** @code{.cogeno.ins}@endcode */
...
```

Lines with `@code{.cogeno.py}` or `@code{.cogeno.ins}@endcode` are marker lines. The lines between `@code{.cogeno.py}` and `@endcode{.cogeno.py}` are the generator Python code. The lines between `@endcode{.cogeno.py}` and `@code{.cogeno.ins}@endcode` are the output from the generator.

When cogeno runs, it discards the last generated Python output, executes the generator Python code, and writes its generated output into the file. All text lines outside of the special markers are passed through unchanged.

The cogeno marker lines can contain any text in addition to the marker tokens. This makes it possible to hide the generator Python code from the source file.

In the sample above, the entire chunk of Python code is a C comment, so the Python code can be left in place while the file is treated as C code.

1.1.2 About cogeno modules

Cogeno includes several modules to support specific code generation tasks.

- cogeno_edts

  The database provides access to the device tree specification data of a project.

  The EDTS database may be loaded from a json file, stored to a json file or extracted from the DTS files and the bindings yaml files of the project. The EDTS database is automatically available to cogeno scripts. It can also be used as a standalone tool.

1.1.3 About the documentation

This documentation is continuously written. It is edited via text files in the reStructuredText markup language and then compiled into a static website/ offline document using the open source Sphinx and Read the Docs tools.

You can contribute to cogeno's dcoumentation by opening GitLab issues or sending patches via merge requests on its GitLab repository.

1.2 Frequently asked questions

TBD

CHAPTER

TWO

GETTING STARTED

2.1 Invoking cogeno

2.1.1 Synopsis

cogeno [OPTIONS]

2.1.2 Description

Cogeno transforms files in a very simple way: it finds chunks of script code embedded in them, executes the script code, and places its output combined with the original file content into the generated file. It supports Python and Jinja2 scripts.

2.1.3 Options

The following options are understood:

-h, --help show this help message and exit

-x, --delete-code Delete the generator code from the output file.

-w, --warn-empty Warn if a file has no generator code in it.

-n ENCODING, --encoding ENCODING Use ENCODING when reading and writing files.

-U, --unix-newlines Write the output with Unix newlines (only LF line-endings).

-D DEFINE, --define DEFINE Define a global string available to your generator code.

-m DIR [DIR ...], --modules DIR [DIR ...] Use modules from modules DIR. We allow multiple

-t DIR [DIR ...], --templates DIR [DIR ...] Use templates from templates DIR. We allow multiple

-c FILE, --config FILE Use configuration variables from configuration FILE.

-k FILE, --cmakecache FILE Use CMake variables from CMake cache FILE.

-d FILE, --dts FILE Load the device tree specification from FILE.

-b DIR [DIR ...], --bindings DIR [DIR ...] Use bindings from bindings DIR for device tree extraction. We allow multiple.

-e FILE, --edts FILE Write or read EDTS database to/ from FILE.

-i FILE, --input FILE Get the input from FILE.

-o FILE, --output FILE Write the output to FILE.

-l FILE, --log FILE Log to FILE.

-k FILE, --lock FILE Use lock FILE for concurrent runs of cogeno.

2.2 Invoking edtsdatabase

2.2.1 Synopsis

edtsdatabase [OPTIONS]

2.2.2 Description

The Extended Device Tree Specification database collates device tree (dts) information with information taken from the device tree bindings.

The EDTS database may be loaded from a json file, stored to a json file or extracted from the DTS files and the bindings yaml files.

2.2.3 Options

The following options are understood:

-h, --help show this help message and exit

-l FILE, --load FILE Load the input from FILE.

-s FILE, --save FILE Save the database to FILE.

-e FILE, --extract FILE Extract the database from dts FILE.

-b DIR [DIR ...], --bindings DIR [DIR ...] Use bindings from bindings DIR for extraction. We allow multiple.

-p, --print Print EDTS database content.

2.3 Scripting

2.3.1 Code generation functions

A module called `cogeno` provides the core functions for inline code generation. It encapsulates all the functions to retrieve information (options, device tree properties, CMake variables, config properties) and to output the generated code.

The `cogeno` module is automatically imported by all code snippets. No explicit import is necessary.

Note: The `cogeno` module provides the public functions of the code generator 'mixin' classes as cogeno functions. You can simply write:

cogeno.func(...)

The mixin class function cogeno.xxx.XxxMixin.func(self, ...) is not directly available to code snippets.

Output

cogeno.out(sOut='', dedent=False, trimblanklines=False)

`cogeno.output.OutputMixin.`**`out`**(*self self*, *sOut sOut = ''*, *dedent dedent = False*, *trimblanklines trimblanklines = False*)

Write text to the output.

`dedent` and `trimblanklines` make it easier to use multi-line strings, and they are only are useful for multi-line strings:

Parameters

- `sOut`: The string to write to the output.
- `dedent`: If dedent is True, then common initial white space is removed from the lines in sOut before adding them to the output.
- `trimblanklines`: If trimblanklines is True, then an initial and trailing blank line are removed from sOut before adding them to the output.

```
cogeno.out("""
   These are lines I
   want to write into my source file.
""", dedent=True, trimblanklines=True)
```

cogeno.outl(sOut='', dedent=False, trimblanklines=False)

`cogeno.output.OutputMixin.`**`outl`**(*self self*, *sOut sOut = ''*, *dedent dedent = False*, *trimblanklines trimblanklines = False*)

Write text to the output with newline appended.

See *OutputMixin::out*(self, sOut='', dedent=False, trimblanklines=False)

Parameters

- `sOut`: The string to write to the output.
- `dedent`: If dedent is True, then common initial white space is removed from the lines in sOut before adding them to the output.
- `trimblanklines`: If trimblanklines is True, then an initial and trailing blank line are removed from sOut before adding them to the output.

The cogeno module also provides a set of convenience functions:

Code generator

cogeno.cogeno_state()

cogeno.generator.CodeGenerator.**cogeno_state**(*self self*)
numeric cogeno state id

Code generation module import

cogeno.import_module(name)

cogeno.importmodule.ImportMixin.**import_module**(*self self*, *name name*)
Import a Cogeno module.

Import a module from the cogeno/modules package.

Parameters

- name: Module to import. Specified without any path.

See *Code generation modules* for the available modules.

Template file inclusion

cogeno.out_include(include_file)

cogeno.include.IncludeMixin.**out_include**(*self self*, *include_file include_file*)
Write the text from include_file to the output.

The include_file is processed by cogeno. Inline code generation in include_file can access the globals defined in the including source file before inclusion. The including source file can access the globals defined in the include_file (after inclusion).

Parameters

- include_file: Path of include file, either absolute path or relative to current directory or relative to templates directory (e.g. 'templates/drivers/simple_tmpl.c')

cogeno.guard_include()

cogeno.include.IncludeMixin.**guard_include**(*self self*)
Prevent the current file to be included by cogeno.out_include() when called the next time.

Error handling

cogeno.error(msg [, frame_index=0] [, snippet_lineno=0])

cogeno.error.ErrorMixin.**error**(*self self*, *msg msg = ' Error raised by cogeno generator.'*, *frame_index frame_index = 0*, *lineno lineno = 0*)

Raise Error exception.

Extra information is added that maps the python snippet line seen by the Python interpreter to the line of the file that inlines the python snippet.

Parameters

- `msg`: [optional] exception message
- `frame_index`: [optional] Call frame index. The call frame offset of the function calling *error()*. Zero if directly called in a snippet. Add one for every level of function call.
- `lineno`: [optional] line number within template

Log output

cogeno.log(message, message_type=None, end="n", logonly=True)

cogeno.log.LogMixin.**log**(*self self*, *message message*, *message_type message_type = None*)

Print message and write to log file.

Parameters

- `message`: Message
- `message_type`: If given will be prepended to the message
- `end`: Character to put at the end of the message. '\n' by default.
- `logonly`: Only write to logfile. True by default.

cogeno.msg(message)

cogeno.log.LogMixin.**msg**(*self self*, *message message*)

Print message to stdout and log with a "message: " prefix.

See *LogMixin::log()*

Parameters

- `message`: Message

cogeno.warning(message)

cogeno.log.LogMixin.**warning**(*self self*, *message message*)
Print message to stdout and log with a "warning: " prefix.

See *LogMixin::log()*

Parameters

- message: Message

cogeno.prout(message, end="n")

cogeno.log.LogMixin.**prout**(*self self*, *message message*)
Print message to stdout and log.

See *LogMixin::log()*

Parameters

- message: Message
- end: Character to put at the end of the message. '\n' by default.

cogeno.prerr(message, end="n")

cogeno.log.LogMixin.**prerr**(*self self*, *message message*)
Print message to stderr and log with a "error: " prefix.

See *LogMixin::log()*

Parameters

- message: Message
- end: Character to put at the end of the message. '\n' by default.

See *Invoking cogeno* for how to provide the path to the file used for logging.

Lock access

cogeno.lock()

cogeno.lock.LockMixin.**lock**(*self self*)
Get the global cogeno lock.

```
try:
     with cogeno.lock().acquire(timeout = 10):
          ...
except cogeno.lock_timeout():
    cogeno.error(...)
except:
    raise
```

Return Lock object

cogeno.lock_timeout()

cogeno.lock.LockMixin.**lock_timeout**(*self self*)

Lock timeout.

Return Lock timeout object

See *Invoking cogeno* for how to provide the path to the file used for locking.

Lock object

cogeno.lock.BaseFileLock.**acquire**(*self self*, *timeout timeout = None*, *poll_intervall poll_intervall = 0.05*)

```
.. code-block:: python

    # You can use this method in the context manager (recommended)
    with lock.acquire():
pass

    # Or use an equivalent try-finally construct:
    lock.acquire()
    try:
pass
    finally:
lock.release()

:arg float timeout:
    The maximum time waited for the file lock.
    If ``timeout <= 0``, there is no timeout and this method will
    block until the lock could be acquired.
    If ``timeout`` is None, the default :attr:`~timeout` is used.

:arg float poll_intervall:
    We check once in *poll_intervall* seconds if we can acquire the
    file lock.

:raises Timeout:
    if the lock could not be acquired in *timeout* seconds.

.. versionchanged:: 2.0.0

    This method returns now a *proxy* object instead of *self*,
    so that it can be used in a with statement without side effects.
```

cogeno.lock.BaseFileLock.**release**(*self self*, *force force = False*)

```
Please note, that the lock is only completely released, if the lock
counter is 0.

Also note, that the lock file itself is not automatically deleted.

:arg bool force:
    If true, the lock counter is ignored and the lock is released in
    every case.
```

cogeno.filelock.BaseFileLock.**is_locked**(*self self*)

True if this lock is currently held.

.. versionchanged:: 2.0.0

This was previously a method and is now a property.

Options

cogeno.option()

cogeno.options.OptionsMixin.**option**(*self self*, *name name*)

Get option of actual context.

Return option value

Parameters

- name: Name of option

cogeno.options_add_argument(*args, **kwargs)

cogeno.options.OptionsMixin.**options_add_argument**(*self self*, *args args*, *kwargs kwargs*)

Add option arguments to option parser of actual context.

Cogeno modules may add arguments to the cogeno option parser. The argument variables given to cogeno are rescanned after new option arguments are provided.

```
def mymodule(cogeno):
    if not hasattr(cogeno, '_mymodule'):
        cogeno._mymodule = None

        cogeno.options_add_argument('-m', '--mymodule', metavar='FILE',
            dest='mymodule_file', action='store',
            type=lambda x: cogeno.options_is_valid_file(x),
            help='Load mymodule data from FILE.')

   if getattr(cogeno, '_mymodule') is not None:
       return cogeno._mymodule

   if cogeno.option('mymodule_file'):
       mymodule_file = cogeno.option('mymodule_file')
   else:
       cogeno.error(..., 2)

   ...
   cogeno._mymodule = ...
```

Path functions

cogeno.path_walk(top, topdown = False, followlinks = False)

cogeno.paths.PathsMixin.**path_walk**(*top top*, *topdown topdown = False*, *followlinks followlinks = False*)

Walk directory tree.

See Python docs for os.walk, exact same behavior but it yields Path() instances instead

From: http://ominian.com/2016/03/29/os-walk-for-pathlib-path/

cogeno.find_template_files(top, marker, suffix='.c')

cogeno.paths.PathsMixin.**find_template_files**(*top top*, *marker marker*, *suffix suffix = '.c'*)

Find template files.

Return List of template file pathes

Parameters

- marker: Marker as b'my-marker'
- suffix:

Standard streams

cogeno.set_standard_streams(self, stdout=None, stderr=None)

cogeno.redirectable.RedirectableMixin.**set_standard_streams**(*self self*, *stdout stdout = None*, *stderr stderr = None*)

Redirect status and error reporting.

Assign new files for standard out and/or standard error.

Parameters

- stdout:
- stderr:

Standard Modules - CMake

cogeno.cmake_variable(variable_name [, default="<unset>"])

cogeno.stdmodules.StdModulesMixin.**cmake_variable**(*self self*, *variable_name variable_name*, *default default = "<unset>"*)

Get the value of a CMake variable.

If variable_name is not provided to cogeno by CMake the default value is returned.

A typical set of CMake variables that are not available in the CMakeCache.txt file and have to be provided as defines to cogeno if needed:

- "PROJECT_NAME"
- "PROJECT_SOURCE_DIR"
- "PROJECT_BINARY_DIR"
- "CMAKE_SOURCE_DIR"
- "CMAKE_BINARY_DIR"
- "CMAKE_CURRENT_SOURCE_DIR"
- "CMAKE_CURRENT_BINARY_DIR"
- "CMAKE_CURRENT_LIST_DIR"
- "CMAKE_FILES_DIRECTORY"
- "CMAKE_PROJECT_NAME"
- "CMAKE_SYSTEM"
- "CMAKE_SYSTEM_NAME"
- "CMAKE_SYSTEM_VERSION"
- "CMAKE_SYSTEM_PROCESSOR"
- "CMAKE_C_COMPILER"
- "CMAKE_CXX_COMPILER"
- "CMAKE_COMPILER_IS_GNUCC"
- "CMAKE_COMPILER_IS_GNUCXX"

Return value

Parameters

- `variable_name`: Name of the CMake variable
- `default`: Default value

cogeno.cmake_cache_variable(variable_name [, default="<unset>"])

`cogeno.stdmodules.StdModulesMixin.`**`cmake_cache_variable`**(*self self*, *variable_name variable_name*, *default default = "<unset>"*)

Get the value of a CMake variable from CMakeCache.txt.

If variable_name is not given in `CMakeCache.txt` the default value is returned.

Return value

Parameters

- `variable_name`: Name of the CMake variable
- `default`: Default value

See *Invoking cogeno* and *Integration into the build process* for how to provide CMake variables to cogeno.

Standard Modules - config

cogeno.config_properties()

cogeno.stdmodules.StdModulesMixin.**config_properties**(*self self*)

Get all config properties.

The property names are the ones config file.

Return A dictionary of config properties.

cogeno.config_property(property_name [, default="<unset>"])

cogeno.stdmodules.StdModulesMixin.**config_property**(*self self*, *property_name property_name*, *default default = "<unset>"*)

Get the value of a configuration property fromthe config file.

If property_name is not given in .config the default value is returned.

Return property value

Parameters

- `property_name`: Name of the property
- `default`: Property value to return per default.

See *Invoking cogeno* and *Integration into the build process* for how to provide config variables to cogeno.

Standard Modules - Extended Device Tree Database

cogeno.edts()

cogeno.stdmodules.StdModulesMixin.**edts**(*self self*)

Get the extended device tree database.

Return Extended device tree database.

See *Invoking cogeno* and *Integration into the build process* for how to provide all files to enable cogeno to build the extended device tree database.

2.3.2 Code generation modules

Code generation modules provide supporting functions for code generation.

Some modules have to be imported to gain access to the module's functions and variables. The standard modules are accessible by convenience functions.

```
/* This file uses modules. */
...
/**
 * @code{.cogeno.py}
 * cogeno.import_module('my_special_module')
```

(continues on next page)

(continued from previous page)

```
 * my_special_module.do_everything():
 * @endcode{.cogeno.py}
 */
/** @code{.cogeno.ins}@endcode */
...
```


Standard modules

Extended device tree specification (EDTS) database

The EDTS database module extracts device tree information from the device tree specification.

THE EDTS database is a key value store. The keys are pathes to the device tree information.

You may get access to the database by `cogeno.edts()`.

In case you want to use the extended device tree database in another Python project import it by:

```
import cogeno.modules.edtsdatabase
```

EDTS bindings

The EDTS database module uses bindings (a kind of data schema) to know what data to extract and to know the kind of data. A set of generic bindings controls the extraction process. The generic bindings are part of the EDTS database module.

Fixed partition

```
include: fixed-partition.yaml
```

```
compatible: "fixed-partition"

properties:
    label:
        type: string
```

(continues on next page)

(continued from previous page)

```
        required: false
        description: The label / name for this partition.  If omitted, the label is␣
↪taken
                     from the node name (excluding the unit address).

    read-only:
        type: boolean
        required: false
        description: This parameter, if present, is a hint that this
                     partition should/ can only be used read-only.

    reg:
        type: array
        required: false
        description: partition offset (address) and size within flash
```

Flash

```
include: flash.yaml
```

```
include: base.yaml

properties:
    label:
      required: true

    reg:
      required: true

    "#address-cells":
      type: int
      required: true
      description: >
        <1>: for flash devices that require a single 32-bit cell to represent their
            address (aka the value is below 4 GiB)
        <2>: for flash devices that require two 32-bit cells to represent their
            address (aka the value is 4 GiB or greater).

    "#size-cells":
      type: int
      required: true
      description: >
        <1>: for flash devices that require a single 32-bit cell to represent their
            size (aka the value is below 4 GiB)
        <2>: for flash devices that require two 32-bit cells to represent their
            size (aka the value is 4 GiB or greater).

    write-block-size:
      type: int
      required: false
      description: Size of flash blocks  for write operations

    erase-block-size:
      type: int
```

(continues on next page)

(continued from previous page)

```
      required: false
      description: Size of flash blocks for erase operations
```

Flash controller

```
include: flash-controller.yaml
```

```
    label:
      required: true

    reg:
      required: true
```

Partitions

```
include: partition.yaml
```

```
compatible: "fixed-partitions"

properties:
    "#address-cells":
      type: int
      required: true
      description: >
        <1>: for partitions that require a single 32-bit cell to represent their
            size/address (aka the value is below 4 GiB)
        <2>: for partitions that require two 32-bit cells to represent their
            size/address (aka the value is 4 GiB or greater).

    "#size-cells":
      type: int
      required: false
      description: >
        <1>: for partitions that require a single 32-bit cell to represent their
            size/address (aka the value is below 4 GiB)
        <2>: for partitions that require two 32-bit cells to represent their
            size/address (aka the value is 4 GiB or greater).
```

SoC non-volatile flash

```
include: soc-nv-flash.yaml
```

```
properties:
    label:
      required: false

    erase-block-size:
     type: int
     description: address alignment required by flash erase operations
```

(continues on next page)

(continued from previous page)

```
    required: false

  write-block-size:
    type: int
    description: address alignment required by flash write operations
    required: false
```

CMake support functions

The CMake module provides access to CMake variables and the CMake cache.

You may get access to the database by `cogeno.stdmodules.StdModulesMixin.cmake()`.

There are convenience functions to access the CMake variables:

- `cogeno.cmake_variable()`
- `cogeno.cmake_cache_variable()`

Other modules

C code generation functions

The ccode module supports code generation for the C language.

To use the module in inline code generation import it by:

```
cogeno.import_module('ccode')
```

In case you want to use the ccode module in another Python project import it by:

```
import cogeno.modules.ccode
```

`cogeno.modules.ccode.`**`outl_config_guard`**(*property_name property_name*)
Write #if guard for config property to output.

If there is a configuration property of the given name the property value is used as guard value, otherwise it is set to 0.

Parameters

- `property_name`: Property name

`cogeno.modules.ccode.`**`outl_config_unguard`**(*property_name property_name*)
Write #endif guard for config property to output.

This is the closing command for outl_guard_config().

Parameters

- `property_name`: Property name

`cogeno.modules.ccode.`**`out_comment`**(*s s*, *blank_before blank_before = True*)
Write 's' as a comment.

Parameters

- `s`: string, is allowed to have multiple lines.
- `blank_before`: True adds a blank line before the comment.

`cogeno.modules.ccode.`**`outl_edts_defines`**(*prefix prefix = 'EDT_'*)

Write EDTS database properties as C defines.

Parameters

- `prefix`: Define label prefix. Default is 'EDT_'.

Zephyr support functions

The Zephyr module supports code generation for the Zephyr RTOS.

To use the module in inline code generation import it by:

```
cogeno.import_module('zephyr')
```

In case you want to use the Zephyr module in another Python project import it by:

```
import cogeno.modules.zephyr
```

Zephyr device declaration

The Zephyr module provides functions to generate device driver instantiations.

```
cogeno.import_module('zephyr')
```

The device delaration functions generate device instances code for all devices activated ('status' = 'ok') in the board device tree file matching the provided compatibles.

Most of the parameters aim at filling the DEVICE_AND_API_INIT macro. Other parameters are there to help code generation to fit driver specifics.

Instance code will only be generated if the Kconfig variable is set. The variable name is build with the device node label name (e.g: CONFIG_I2C_1).

Driver info templates

The device declaration functions work on templates that feature placeholder substitution.

Device instance property placeholders:

- **${device-name}: device instance name.** Name is generated by the declaration function.
- **${driver-name}: device instance driver name.** Name is taken from the device tree node property 'label'.
- **${device-data}: device instance data structure name.** Name is generated by the declaration function.
- **${device-config-info}: device instance configuration structure name.** Name is generated by the declaration function.
- **${device-config-irq}: device instance interrupt configuration function name.** Name is generated by the declaration function.

Device instance device tree property placeholders:

- **${[path to DTS property]}: device instance device tree node property.** The property path supports every node property that is documented in the node yaml bindings. It also supports yaml heuristics, like 'bus-master' and will use documented '"#cells"'.

Device tree property placeholders:

- **${[device id]:[path to DTS property]}: device node property value.** The device node property is defined by the property path of the device given by the device id. The device id is usually also taken from a DTS property e.g. ${${clock/0/controller}:device-name}.

KConfig configuration parameter placeholders:

- ${CONFIG_[configuration parameter]}: KConfig configuration parameter value.

Example:

'c' template code between triple quotes ("""" """") that should provide the expected code to be generated for the driver structures.

```
"""
#if CONFIG_SPI_STM32_INTERRUPT
DEVICE_DECLARE(${device-name});
static void ${device-config-irq}(struct device *dev)
{
        IRQ_CONNECT(${interrupts/0/irq}, ${interrupts/0/priority}, \\
                    spi_stm32_isr, \\
                    DEVICE_GET(${device-name}), 0);
        irq_enable(${interrupts/0/irq});
}
#endif
static const struct spi_stm32_config ${device-config-info} = {
        .spi = (SPI_TypeDef *)${reg/0/address/0},
        .pclken.bus = ${clocks/0/bus},
        .pclken.enr = ${clocks/0/bits},
#if CONFIG_SPI_STM32_INTERRUPT
        .config_irq = ${device-config-irq},
#endif
};
static struct spi_stm32_data ${device-data} = {
        SPI_CONTEXT_INIT_LOCK(${device-data}, ctx),
        SPI_CONTEXT_INIT_SYNC(${device-data}, ctx),
};
"""
```

Declaration of a single device instance

zephyr.**device_declare_single**(*device_config*, *driver_name*, *device_init*, *device_pm_control*, *device_level*, *device_prio*, *device_api*, *device_info*)

Generate device instance code for a device instance that:

- matches the driver name that
- is activated ('status' = 'ok') in the board device tree file and that is
- configured by Kconfig.

Parameters

- **device_config** – Configuration variables for device instantiation. (e.g. 'CONFIG_SPI_0')

- **driver_name** – Driver name for device instantiation. (e.g. 'SPI_0')
- **device_init** – Device initialisation function. (e.g. 'spi_stm32_init')
- **device_level** – Driver initialisation level. (e.g. 'PRE_KERNEL_1')
- **device_prios** – Driver initialisation priority definition. (e.g. 32)
- **device_api** – Identifier of the device api. (e.g. 'spi_stm32_driver_api')
- **device_info** – Device info template for device driver config, data and interrupt initialisation.
- **device_defaults** – Default property values. (e.g. { 'label' : 'My default label' })

Param device_pm_control: Device power management function. (e.g. 'device_pm_control_nop')

Declaration of multiple device instances

zephyr.**device_declare_multi**(*device_configs*, *driver_names*, *device_inits*, *device_levels*, *device_prios*, *device_api*, *device_info*)

Generate device instances code for all device instances that:

- match the driver names that
- are activated ('status' = 'ok') in the board device tree file and that are
- configured by Kconfig.

Parameters

- **device_configs** – A list of configuration variables for device instantiation. (e.g. ['CONFIG_SPI_0', 'CONFIG_SPI_1'])
- **driver_names** – A list of driver names for device instantiation. The list shall be ordered the same way as the list of device configs. (e.g. ['SPI_0', 'SPI_1'])
- **device_inits** – A list of device initialisation functions or a single function. The list shall be ordered as the list of device configs. (e.g. 'spi_stm32_init')
- **device_levels** – A list of driver initialisation levels or a single level definition. The list shall be ordered as the list of device configs. (e.g. 'PRE_KERNEL_1')
- **device_prios** – A list of driver initialisation priorities or a single priority definition. The list shall be ordered as the list of device configs. (e.g. 32)
- **device_api** – Identifier of the device api. (e.g. 'spi_stm32_driver_api')
- **device_info** – Device info template for device driver config, data and interrupt initialisation.
- **device_defaults** – Default property values. (e.g. { 'label' : 'My default label' })

Param device_pm_controls: A list of device power management functions or a single function. The list shall be ordered as the list of device configs. (e.g. 'device_pm_control_nop')

Example:

```
/**
 * @code{.cogeno}
 * cogeno.import_module('zephyr')
 *
```

(continues on next page)

(continued from previous page)

```
 * device_configs = ['CONFIG_SPI_{}'.format(x) for x in range(1, 4)]
 * driver_names = ['SPI_{}'.format(x) for x in range(1, 4)]
 * device_inits = 'spi_stm32_init'
 * device_pm_controls = 'device_pm_control_nop'
 * device_levels = 'POST_KERNEL'
 * device_prios = 'CONFIG_SPI_INIT_PRIORITY'
 * device_api = 'spi_stm32_driver_api'
 * device_info = \
 * """
 * #if CONFIG_SPI_STM32_INTERRUPT
 * DEVICE_DECLARE(${device-name});
 * static void ${device-config-irq}(struct device *dev)
 * {
 *         IRQ_CONNECT(${interrupts/0/irq}, ${interrupts/0/priority}, \\
 *                     spi_stm32_isr, \\
 *                     DEVICE_GET(${device-name}), 0);
 *         irq_enable(${interrupts/0/irq});
 * }
 * #endif
 * static const struct spi_stm32_config ${device-config-info} = {
 *         .spi = (SPI_TypeDef *)${reg/0/address/0},
 *         .pclken.bus = ${clocks/0/bus},
 *         .pclken.enr = ${clocks/0/bits},
 * #if CONFIG_SPI_STM32_INTERRUPT
 *         .config_irq = ${device-config-irq},
 * #endif
 * };
 * static struct spi_stm32_data ${device-data} = {
 *         SPI_CONTEXT_INIT_LOCK(${device-data}, ctx),
 *         SPI_CONTEXT_INIT_SYNC(${device-data}, ctx),
 * };
 * """
 *
 * zephyr.device_declare_multi( \
 *     device_configs,
 *     driver_names,
 *     device_inits,
 *     device_pm_controls,
 *     device_levels,
 *     device_prios,
 *     device_api,
 *     device_info)
 * @endcode{.cogeno}
 */
/** @code{.codeins}@endcode */
```

2.3.3 Code Generation Templates

Code generation templates provide sophisticated code generation functions.

Templates are simply text files. They may be hierarchical organized. There is always one top level template. All the other templates have to be included to gain access to the template's functions and variables.

A template file usually contains normal text and templating commands intermixed. A bound sequence of templating commands is called a script snippet. As a special case a template file may be a script snippet as a whole.

Cogeno supports two flavours of script snippets: Python and Jinja2. A script snippet has to be written in one of the two scripting languages. Within a template file snippets of different language can coexist.

Template Snippets

```
/* This file uses templates. */
...
/**
 * @code{.cogeno.py}
 * template_in_var = 1
 * cogeno.out_include('templates/template_tmpl.c')
 * if template_out_var not None:
 *     cogeno.outl("int x = %s;" % template_out_var)
 * @endcode{.cogeno.py}
 */
/** @code{.cogeno.ins}@endcode */
...
```

2.4 Integration into the build process

Code generation has to be invoked as part of the build process of a project.

2.4.1 CMake

Projects that use CMake to manage building the project can add the following CMake code to the CMake scripts.

By this a file that contains inline code generation can be added to the project using the *target_sources_cogeno* command in the respective `CMakeList.txt` file.

`target_sources_cogeno` (*file [COGENO_DEFINES defines..] [DEPENDS target.. file..]*)

```
find_program(COGENO_EXECUTABLE cogeno)

if(EXISTS "${COGENO_EXECUTABLE}")
    # We do not need the Python 3 interpreter
    set(COGENO_EXECUTABLE_OPTION)
else()
    # Cogeno is not installed.
    # Maybe the cogeno repository was cloned
    # or the Zephyr copy is available.
    find_file(COGENO_PY cogeno.py
              PATHS $ENV{HOME}/cogeno/cogeno
                    $ENV{ZEPHYR_BASE}/../cogeno/cogeno
                    $ENV{ZEPHYR_BASE}/scripts/cogeno/cogeno)

    if(NOT EXISTS "${COGENO_PY}")
        message(FATAL_ERROR "Cogeno not found - '${COGENO_PY}'.")
    endif()

    if(${CMAKE_VERSION} VERSION_LESS "3.12")
        set(Python_ADDITIONAL_VERSIONS 3.7 3.6 3.5)
        find_package(PythonInterp)

        set(Python3_Interpreter_FOUND ${PYTHONINTERP_FOUND})
        set(Python3_EXECUTABLE ${PYTHON_EXECUTABLE})
        set(Python3_VERSION ${PYTHON_VERSION_STRING})
    else()
        # CMake >= 3.12
        find_package(Python3 COMPONENTS Interpreter)
    endif()

    if(NOT ${Python3_Interpreter_FOUND})
        message(FATAL_ERROR "Python 3 not found")
    endif()

    set(COGENO_EXECUTABLE "${Python3_EXECUTABLE}")
    set(COGENO_EXECUTABLE_OPTION "${COGENO_PY}")

    message(STATUS "Found cogeno: '${COGENO_PY}'.")
endif()

# --config
if(COGENO_CONFIG)
    set(COGENO_CONFIG_OPTION "--config" "${COGENO_CONFIG}")
else()
    set(COGENO_CONFIG_OPTION)
endif()

# --dts
# --bindings
# --edts
if(COGENO_DTS)
    set(COGENO_DTS_OPTION "--dts" "${COGENO_DTS}")
    if(COGENO_BINDINGS)
        set(COGENO_BINDINGS_OPTION "--bindings" ${COGENO_BINDINGS})
    else()
        set(COGENO_BINDINGS_OPTION)
    endif()
```

(continues on next page)

(continued from previous page)

```
    if(COGENO_EDTS)
        set(COGENO_EDTS_OPTION "--edts" "${COGENO_EDTS}")
    else()
        set(COGENO_EDTS_OPTION "--edts" "${CMAKE_BINARY_DIR}/edts.json")
    endif()
else()
    set(COGENO_DTS_OPTION)
    set(COGENO_BINDINGS_OPTION)
    if(COGENO_EDTS)
        set(COGENO_EDTS_OPTION "--edts" "${COGENO_EDTS}")
    else()
        set(COGENO_EDTS_OPTION)
    endif()
endif()

# --modules
if(COGENO_MODULES)
    set(COGENO_MODULES_OPTION "--modules" ${COGENO_MODULES})
else()
    set(COGENO_MODULES_OPTION)
endif()

# --modules
if(COGENO_TEMPLATES)
    set(COGENO_TEMPLATES_OPTION "--templates" ${COGENO_TEMPLATES})
else()
    set(COGENO_TEMPLATES_OPTION)
endif()

function(target_sources_cogeno
    target          # The CMake target that depends on the generated file
    )

    set(options)
    set(oneValueArgs)
    set(multiValueArgs COGENO_DEFINES DEPENDS)
    cmake_parse_arguments(COGENO "${options}" "${oneValueArgs}"
                          "${multiValueArgs}" ${ARGN})
    # prepend -D to all defines
    string(REGEX REPLACE "([^;]+)" "-D;\\1"
            COGENO_COGENO_DEFINES "${COGENO_COGENO_DEFINES}")

    message(STATUS "Will generate for target ${target}")
    # Generated file must be generated to the current binary directory.
    # Otherwise this would trigger CMake issue #14633:
    # https://gitlab.kitware.com/cmake/cmake/issues/14633
    foreach(arg ${COGENO_UNPARSED_ARGUMENTS})
        if(IS_ABSOLUTE ${arg})
            set(template_file ${arg})
            get_filename_component(generated_file_name ${arg} NAME)
            set(generated_file ${CMAKE_CURRENT_BINARY_DIR}/${generated_file_name})
        else()
            set(template_file ${CMAKE_CURRENT_SOURCE_DIR}/${arg})
            set(generated_file ${CMAKE_CURRENT_BINARY_DIR}/${arg})
        endif()
        get_filename_component(template_dir ${template_file} DIRECTORY)
        get_filename_component(generated_dir ${generated_file} DIRECTORY)
```

(continues on next page)

(continued from previous page)

```
    if(IS_DIRECTORY ${template_file})
        message(FATAL_ERROR "target_sources_cogeno() was called on a directory")
    endif()

    # Generate file from template
    message(STATUS " from '${template_file}'")
    message(STATUS " to   '${generated_file}'")
    add_custom_command(
        COMMENT "cogeno ${generated_file}"
        OUTPUT ${generated_file}
        MAIN_DEPENDENCY ${template_file}
        DEPENDS
        ${COGENO_DEPENDS}
        COMMAND
        ${COGENO_EXECUTABLE}
        ${COGENO_EXECUTABLE_OPTION}
        ${COGENO_COGENO_DEFINES}
        -D "\"APPLICATION_SOURCE_DIR=${APPLICATION_SOURCE_DIR}\""
        -D "\"APPLICATION_BINARY_DIR=${APPLICATION_BINARY_DIR}\""
        -D "\"PROJECT_NAME=${PROJECT_NAME}\""
        -D "\"PROJECT_SOURCE_DIR=${PROJECT_SOURCE_DIR}\""
        -D "\"PROJECT_BINARY_DIR=${PROJECT_BINARY_DIR}\""
        -D "\"CMAKE_SOURCE_DIR=${CMAKE_SOURCE_DIR}\""
        -D "\"CMAKE_BINARY_DIR=${CMAKE_BINARY_DIR}\""
        -D "\"CMAKE_CURRENT_SOURCE_DIR=${CMAKE_CURRENT_SOURCE_DIR}\""
        -D "\"CMAKE_CURRENT_BINARY_DIR=${CMAKE_CURRENT_BINARY_DIR}\""
        -D "\"CMAKE_CURRENT_LIST_DIR=${CMAKE_CURRENT_LIST_DIR}\""
        -D "\"CMAKE_FILES_DIRECTORY=${CMAKE_FILES_DIRECTORY}\""
        -D "\"CMAKE_PROJECT_NAME=${CMAKE_PROJECT_NAME}\""
        -D "\"CMAKE_SYSTEM=${CMAKE_SYSTEM}\""
        -D "\"CMAKE_SYSTEM_NAME=${CMAKE_SYSTEM_NAME}\""
        -D "\"CMAKE_SYSTEM_VERSION=${CMAKE_SYSTEM_VERSION}\""
        -D "\"CMAKE_SYSTEM_PROCESSOR=${CMAKE_SYSTEM_PROCESSOR}\""
        -D "\"CMAKE_C_COMPILER=${CMAKE_C_COMPILER}\""
        -D "\"CMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}\""
        -D "\"CMAKE_COMPILER_IS_GNUCC=${CMAKE_COMPILER_IS_GNUCC}\""
        -D "\"CMAKE_COMPILER_IS_GNUCXX=${CMAKE_COMPILER_IS_GNUCXX}\""
        --cmakecache "${CMAKE_BINARY_DIR}/CMakeCache.txt"
        ${COGENO_CONFIG_OPTION}
        ${COGENO_DTS_OPTION}
        ${COGENO_BINDINGS_OPTION}
        ${COGENO_EDTS_OPTION}
        ${COGENO_MODULES_OPTION}
        ${COGENO_TEMPLATES_OPTION}
        --input "${template_file}"
        --output "${generated_file}"
        --log "${CMAKE_BINARY_DIR}${CMAKE_FILES_DIRECTORY}/cogeno.log"
        --lock "${CMAKE_BINARY_DIR}${CMAKE_FILES_DIRECTORY}/cogeno.lock"
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
    )
    target_sources(${target} PRIVATE ${generated_file})
    # Add template directory to include path to allow includes with
    # relative path in generated file to work
    target_include_directories(${target} PRIVATE ${template_dir})
    # Add directory of generated file to include path to allow includes
    # of generated header file with relative path.
```

(continues on next page)

(continued from previous page)

```
        target_include_directories(${target} PRIVATE ${generated_dir})
    endforeach()
endfunction()
```

2.4.2 Zephyr

Cogeno can be integrated into Zephyr by applying the cogeno pull request.

In Zephyr the processing of source files is controlled by the CMake extension functions: `zephyr_sources_cogeno(..)` or `zephyr_library_sources_cogeno(..)`. The generated source files are added to the Zephyr sources. During build the source files are processed by cogeno and the generated source files are written to the CMake binary directory. Zephyr uses CMake as the tool to manage building the project. A file that contains inline code generation has to be added to the project by one of the following commands in a `CMakeList.txt` file:

zephyr_sources_cogeno (*file [COGENO_DEFINES defines..] [DEPENDS target.. file..]*)

zephyr_sources_cogeno_ifdef (*ifguard file [COGENO_DEFINES defines..] [DEPENDS target.. file..]*)

zephyr_library_sources_cogeno (*file [COGENO_DEFINES defines..] [DEPENDS target.. file..]*)

zephyr_library_sources_cogeno_ifdef (*ifguard file [COGENO_DEFINES defines..] [DEPENDS target.. file..]*)

The arguments given by the `COGENO_DEFINES` keyword have to be of the form `define_name=define_value`. The arguments become globals in the python snippets and can be accessed by `define_name`.

Dependencies given by the `DEPENDS` key word are added to the dependencies of the generated file.

CHAPTER

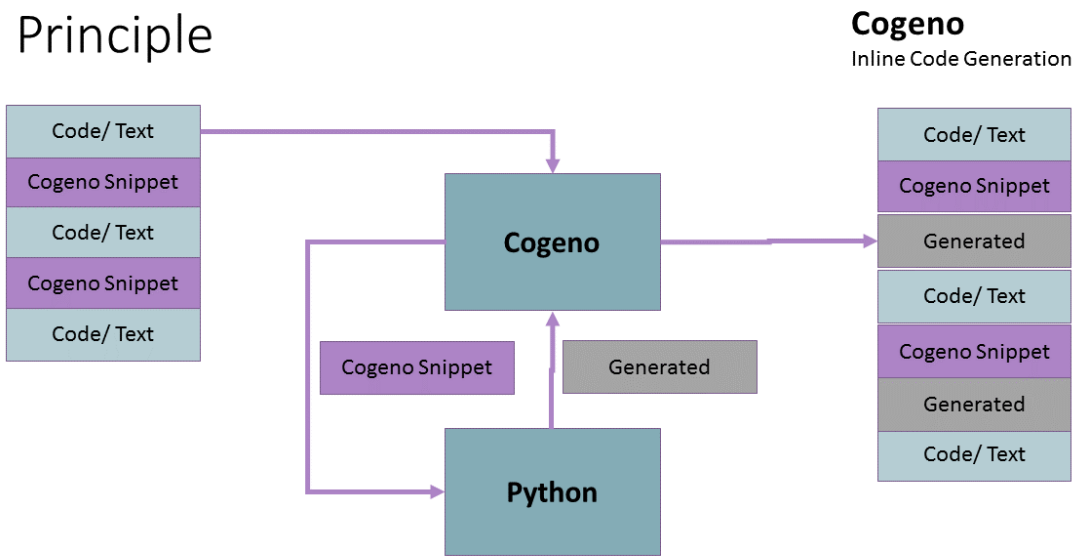
THREE

DEVELOPMENT

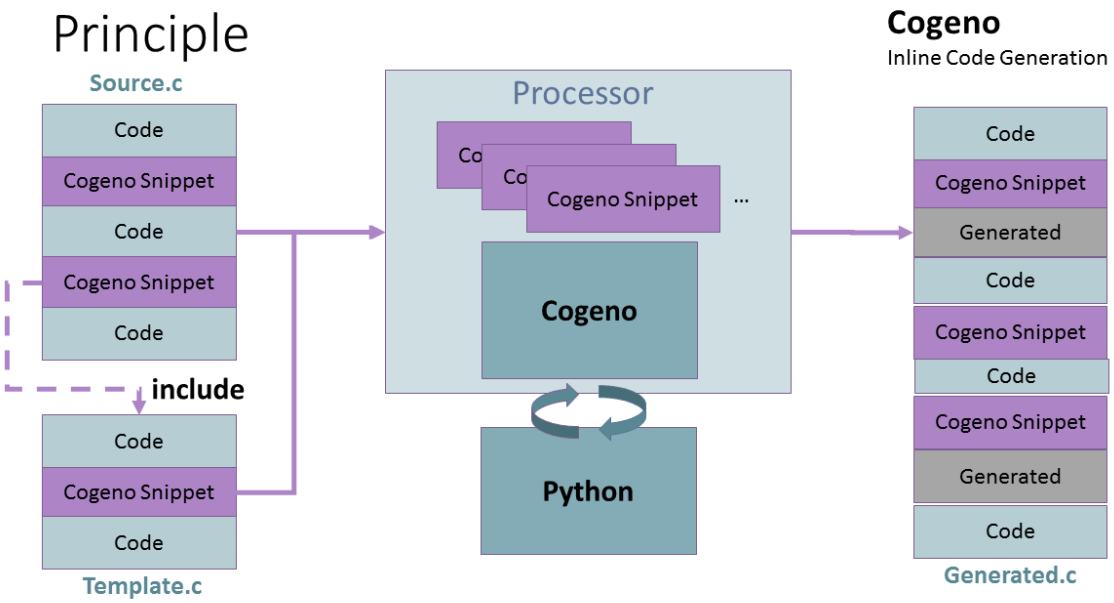
3.1 Code generation principle

How code generation works with cogeno.

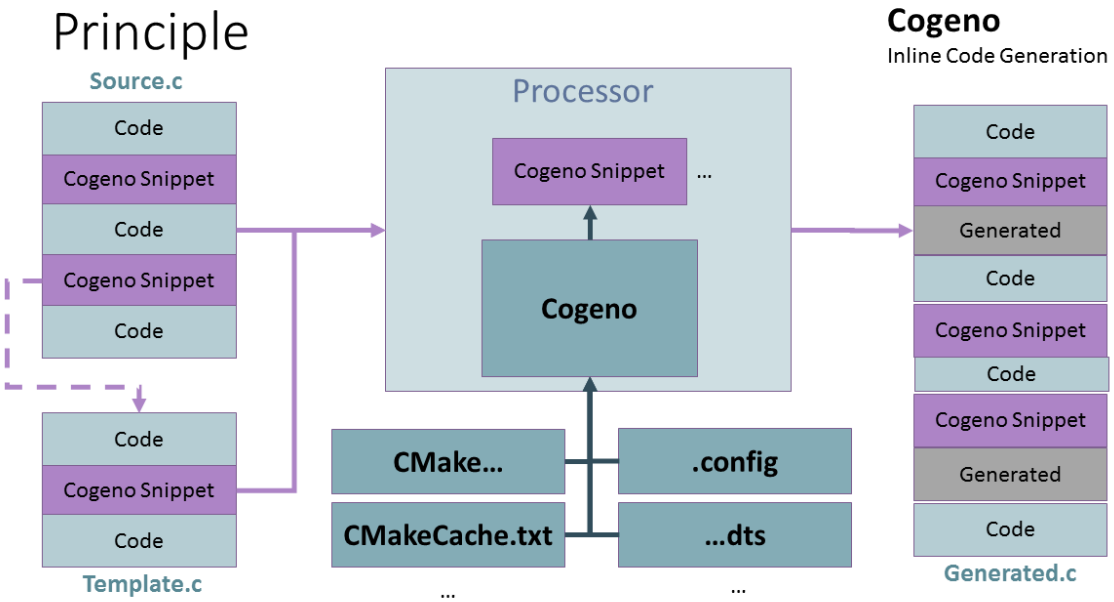
3.1.1 Principle



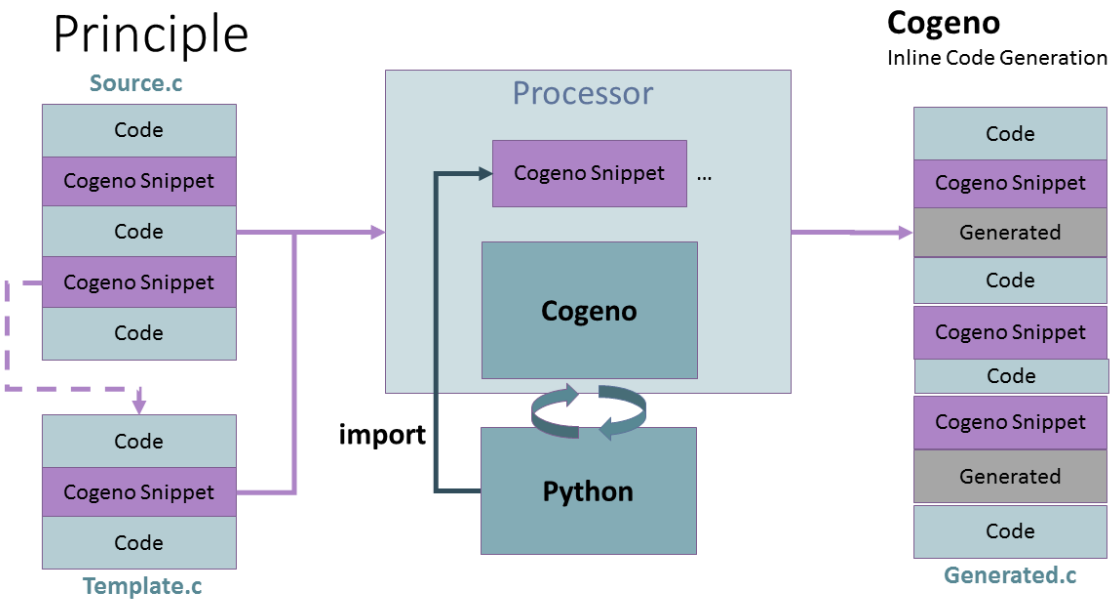
3.1.2 Inclusion of other inline code



3.1.3 Access to project data



3.1.4 Import of Python modules



3.2 Cogeno API

cogeno is a Python module that provides access to the public functions of the class: `CodeGenerator` and the sub-classes of it. See *Code generation functions* for a description of all `cogeno` module's functions.

The interfaces listed hereafter are the internal interface of cogeno.

- *CodeGenerator*
  - *ErrorMixin*
  - *GenericMixin*
  - *LockMixin*
  - *OptionsMixin*
  - *StdModulesMixin*
- *Context*

3.2.1 CodeGenerator

class CodeGenerator

Public Functions

__init__ (*self self*)

cogeno_state (*self self*)
numeric cogeno state id

Public Static Attributes

cogeno.generator.CodeGenerator.cogeno_module = None

list cogeno.generator.CodeGenerator.cogeno_module_states = []

The `CodeGenerator` class includes (sub-classes) several mixin classes:

ErrorMixin

class ErrorMixin
Subclassed by *cogeno.generator.CodeGenerator*

Public Functions

error (*self self*, *msg msg = ' Error raised by cogeno generator.'*, *frame_index frame_index = 0*, *lineno lineno = 0*)
Raise Error exception.

Extra information is added that maps the python snippet line seen by the Python interpreter to the line of the file that inlines the python snippet.

Parameters

- `msg`: [optional] exception message
- `frame_index`: [optional] Call frame index. The call frame offset of the function calling *error()*. Zero if directly called in a snippet. Add one for every level of function call.
- `lineno`: [optional] line number within template

GenericMixin

Warning: doxygenclass: Cannot find class "cogeno::generic::GenericMixin" in doxygen xml output for project "cogeno" from directory: /home/docs/checkouts/readthedocs.org/user_builds/cogeno/checkouts/stable/docs/_build/doxy/xml

LockMixin

class LockMixin
Subclassed by *cogeno.generator.CodeGenerator*

Public Functions

lock_file (*self self*)
Lock file used for the current context.

Return lock file name

lock (*self self*)
Get the global cogeno lock.

```
try:
     with cogeno.lock().acquire(timeout = 10):
         ...
except cogeno.lock_timeout():
    cogeno.error(...)
except:
    raise
```

Return Lock object

lock_timeout (*self self*)
Lock timeout.

Return Lock timeout object

OptionsMixin

class OptionsMixin
Subclassed by *cogeno.generator.CodeGenerator*

Public Functions

option (*self self*, *name name*)
Get option of actual context.

Return option value

Parameters

- `name`: Name of option

options_add_argument (*self self*, *args args*, *kwargs kwargs*)
Add option arguments to option parser of actual context.

Cogeno modules may add arguments to the cogeno option parser. The argument variables given to cogeno are rescanned after new option arguments are provided.

```
def mymodule(cogeno):
    if not hasattr(cogeno, '_mymodule'):
        cogeno._mymodule = None

        cogeno.options_add_argument('-m', '--mymodule', metavar='FILE',
            dest='mymodule_file', action='store',
            type=lambda x: cogeno.options_is_valid_file(x),
```

(continues on next page)

(continued from previous page)

```
            help='Load mymodule data from FILE.')

    if getattr(cogeno, '_mymodule') is not None:
        return cogeno._mymodule

    if cogeno.option('mymodule_file'):
        mymodule_file = cogeno.option('mymodule_file')
    else:
        cogeno.error(..., 2)

    ...
    cogeno._mymodule = ...
```

options_is_valid_file (*self self*, *filepath filepath*)

options_is_valid_directory (*self self*, *directorypath directorypath*)

StdModulesMixin

class StdModulesMixin

Subclassed by *cogeno.generator.CodeGenerator*

Public Functions

edts (*self self*)

Get the extended device tree database.

Return Extended device tree database.

cmake (*self self*)

Get the cmake variables database.

Return CMake variables database.

cmake_variable (*self self*, *variable_name variable_name*, *default default = "<unset>"*)

Get the value of a CMake variable.

If variable_name is not provided to cogeno by CMake the default value is returned.

A typical set of CMake variables that are not available in the `CMakeCache.txt` file and have to be provided as defines to cogeno if needed:

- "PROJECT_NAME"
- "PROJECT_SOURCE_DIR"
- "PROJECT_BINARY_DIR"
- "CMAKE_SOURCE_DIR"
- "CMAKE_BINARY_DIR"
- "CMAKE_CURRENT_SOURCE_DIR"
- "CMAKE_CURRENT_BINARY_DIR"
- "CMAKE_CURRENT_LIST_DIR"

- ”CMAKE_FILES_DIRECTORY”
- ”CMAKE_PROJECT_NAME”
- ”CMAKE_SYSTEM”
- ”CMAKE_SYSTEM_NAME”
- ”CMAKE_SYSTEM_VERSION”
- ”CMAKE_SYSTEM_PROCESSOR”
- ”CMAKE_C_COMPILER”
- ”CMAKE_CXX_COMPILER”
- ”CMAKE_COMPILER_IS_GNUCC”
- ”CMAKE_COMPILER_IS_GNUCXX”

Return value

Parameters

- `variable_name`: Name of the CMake variable
- `default`: Default value

cmake_cache_variable(*self self*, *variable_name variable_name*, *default default = "<unset>"*)
Get the value of a CMake variable from CMakeCache.txt.

If variable_name is not given in `CMakeCache.txt` the default value is returned.

Return value

Parameters

- `variable_name`: Name of the CMake variable
- `default`: Default value

config_properties(*self self*)
Get all config properties.

The property names are the ones config file.

Return A dictionary of config properties.

config_property(*self self*, *property_name property_name*, *default default = "<unset>"*)
Get the value of a configuration property fromthe config file.

If property_name is not given in .config the default value is returned.

Return property value

Parameters

- `property_name`: Name of the property
- `default`: Property value to return per default.

3.2.2 Context

class Context

Context for code generation.

Public Functions

__init__ (*self self*, *generator generator*, *parent_context parent_context = None*, *generation_globals generation_globals = None*, *options options = None*, *eval_begin eval_begin = None*, *eval_end eval_end = None*, *eval_adjust eval_adjust = None*, *delete_code delete_code = None*, *template_file template_file = None*, *template template = None*, *template_source_type template_source_type = None*, *script_type script_type = None*, *template_tabsize template_tabsize = None*, *templates_paths templates_paths = None*, *modules_paths modules_paths = None*, *jinja2_environment jinja2_environment = None*, *output_file output_file = None*, *log_file log_file = None*, *lock_file lock_file = None*)

__str__ (*self self*)

__repr__ (*self self*)

parent (*self self*)

generation_globals (*self self*)

script_is_inline (*self self*)

script_is_python (*self self*)

script_is_jinja2 (*self self*)

script_type (*self self*)

template_is_snippet (*self self*)
Template is a snippet.

Snippets are parts of the template of the parent context.

Return True in case the template is a snippet, False otherwise.

template_is_file (*self self*)

template_is_string (*self self*)

out (*self self*, *line line*)
Add line.

outl (*self self*, *line line*)
Add line with newline.

3.3 edtsdb API

edtsdb is a Python module with the primary class: `EDTSDb`.

The basis for the edtsb module is the `edtlib` library.

- *EDTSDb*

- *edtlib*

3.3.1 EDTSDb

class EDTSDb

Extended DTS database.

Database schema:

```
_edts dict(
  'aliases': dict(alias) : sorted list(device-id)),
  'chosen': dict(chosen),
  'devices':  dict(device-id :  device-struct),
  'compatibles':  dict(compatible : sorted list(device-id)),
  ...
)

device-struct dict(
  'device-id' : device-id,
  'compatible' : list(compatible) or compatible,
  'label' : label,
  '<property-name>' : property-value
)

property-value
  for "boolean", "string", "int" -> string
  for 'array', "string-array", "uint8-array" -> dict(index : string)
  for "phandle-array" -> dict(index : device-id)
```

Database types:

- device-id: opaque id for a device (do not use for other purposes),
- compatible: any of [‘st,stm32-spi-fifo’, . . .] - ‘compatibe’ from <binding>.yaml
- label: any of [‘UART_0’, ‘SPI_11’, . . .] - label directive from DTS

Subclassed by cogeno.modules.edtsdatabase.EDTSDatabase

Public Functions

__init__(*self self*, *args args*, *kw kw*)

__getitem__(*self self*, *key key*)

__iter__(*self self*)

__len__(*self self*)

The EDTSDb class includes (sub-classes) several mixin classes:

class EDTSConsumerMixin

ETDS Database consumer.

Methods for ETDS database usage.

Subclassed by *cogeno.modules.edtsdb.database.EDTSDb*

Public Functions

info (*self self*)
Get info.

Return edts 'info' dict

Parameters

- None:

compatibles (*self self*)
Get compatibles.

Return edts 'compatibles' dict

Parameters

- None:

aliases (*self self*)
Get aliases.

Return edts 'aliases' dict

Parameters

- None:

chosen (*self self*)
Get chosen.

Return edts 'chosen' dict

Parameters

- None:

device_ids_by_compatible (*self self*, *compatibles compatibles*)
Get device ids of all activated compatible devices.

Return list of device ids of activated devices that are compatible

Parameters

- compatibles: compatible(s)

device_id_by_name (*self self*, *name name*)
Get device id of activated device with given name.

Return device id

Parameters

- name:

device_name_by_id (*self self*, *device_id device_id*)
Get label/ name of a device by device id.

If the label is omitted, the name is taken from the node name (including unit address).

Return name

Parameters

- `device_id`:

device_property(*self self*, *device_id device_id*, *property_path property_path*, *default default = "<unset>"*)

Get device tree property value of a device.

Return property value

Parameters

- `device_id`:
- `property_path`: Path of the property to access (e.g. 'reg/0', 'interrupts/prio', 'device_id', ...)

device_properties(*self self*, *device_id device_id*)

device_properties_flattened(*self self*, *device_id device_id*, *path_prefix path_prefix = ""*)

Device properties flattened to property path : value.

Return dictionary of property_path and property_value

Parameters

- `device_id`:
- `path_prefix`:

device_template_substitute(*self self*, *device_id device_id*, *template template*, *presets presets = {}*, *aliases aliases = {}*)

Substitude device property value placeholders in template.

Local placeholders may be defined with direct and indirect path resolution:

- ${<property_path>}
- ${path/${<property_path>}}
- ${path/${<device-id>:<property_path>}}

Global placeholders may also be defined with direct and indirect path resolution:

- ${<device-id>:<property_path>}
- ${${<property_path>}:<property_path>}
- ${${path/${<property_path>}}:<property_path>}
- ${${path/${<device-id>:<property_path>}}:<property_path>}
- ${${<device-id>:<property_path>}:<property_path>}
- ${${<device-id>:path/${<property_path>}}:<property_path>}
- ${${<device-id>:path/${<device-id>:<property_path>}}:<property_path>}

Parameters

- `device_id`:
- `template`:

- `presets`: dict of preset property-path : property value items either of the local form "<property_path>" : value or the global form "<device-id>:<property_path>" : value
- `aliases`: dict of property path alias : property path items.

load (*self self*, *file_path file_path*)
Load extended device tree database from JSON file.

Parameters

- `file_path`: Path of JSON file

class _DeviceGlobalTemplate

Public Static Attributes

string cogeno.modules.edtsdb.consumer.EDTSConsumerMixin._DeviceGlobalTemplate.idpat

class _DeviceLocalTemplate

Public Static Attributes

string cogeno.modules.edtsdb.consumer.EDTSConsumerMixin._DeviceLocalTemplate.idpatt

class EDTSExtractorMixin
ETDS Database extractor.

Methods for ETDS database extraction from DTS.

Subclassed by *cogeno.modules.edtsdb.database.EDTSDb*

Public Functions

extract (*self self*, *dts_path dts_path*, *bindings_dirs bindings_dirs*)
Extract DTS info to database.

Parameters

- `dts_path`: DTS file
- `bindings_dirs`: YAML file directories, we allow multiple

dts_path (*self self*)
Device tree file path.

Return Device tree file path

bindings_dirs (*self self*)
Bindings directories paths.

Return List of binding directories

dts_source (*self self*)
DTS source code.

DTS source code of the loaded devicetree after merging nodes and processing /delete-node/ and /delete-property/.

Return DTS source code as string

class EDTSProviderMixin
ETDS Database provider.

Methods for ETDS database creation.

Subclassed by *cogeno.modules.edtsdb.database.EDTSDb*

Public Functions

insert_alias (*self self*, *alias_path alias_path*, *alias_value alias_value*)
Insert an alias.

Parameters

- `alias_path`: Alias
- `alias_value`: The value the alias aliases.

insert_chosen (*self self*, *chosen_path chosen_path*, *chosen_value chosen_value*)

insert_child_property (*self self*, *device_id device_id*, *child_name child_name*, *property_path property_path*, *property_value property_value*)
Insert property value for the child of a device id.

Parameters

- `device_id`:
- `child_name`:
- `property_path`: Path of the property to access (e.g. 'reg/0', 'interrupts/prio', 'label', ...)
- `property_value`: value

insert_device_property (*self self*, *device_id device_id*, *property_path property_path*, *property_value property_value*)
Insert property value for the device of the given device id.

Parameters

- `device_id`:
- `property_path`: Path of the property to access (e.g. 'reg/0', 'interrupts/prio', 'label', ...)
- `property_value`: value

save (*self self*, *file_path file_path*)
Write json file.

Parameters

- `file_path`: Path of the file to write

3.3.2 edtlib

class EDT

```
Represents a devicetree augmented with information from bindings.

These attributes are available on EDT objects:

nodes:
  A list of Node objects for the nodes that appear in the devicetree

compat2enabled:
  A collections.defaultdict that maps each 'compatible' string that appears
  on some enabled Node to a list of enabled Nodes.

  For example, edt.compat2enabled["bar"] would include the 'foo' and 'bar'
  nodes below.

    foo {
            compatible = "bar";
            status = "okay";
            ...
    };
    bar {
            compatible = "foo", "bar", "baz";
            status = "okay";
            ...
    };


dts_path:
  The .dts path passed to __init__()

dts_source:
  The final DTS source code of the loaded devicetree after merging nodes
  and processing /delete-node/ and /delete-property/, as a string

bindings_dirs:
  The bindings directory paths passed to __init__()
```

Public Functions

__init__(*self self*, *dts dts*, *bindings_dirs bindings_dirs*, *warn_file warn_file = None*)

```
EDT constructor. This is the top-level entry point to the library.

dts:
  Path to devicetree .dts file

bindings_dirs:
  List of paths to directories containing bindings, in YAML format.
  These directories are recursively searched for .yaml files.

warn_file:
  'file' object to write warnings to. If None, sys.stderr is used.
```

get_node(*self self*, *path path*)

```
Returns the Node at the DT path or alias 'path'. Raises EDTError if the
path or alias doesn't exist.
```

chosen_node(*self self*, *name name*)

```
Returns the Node pointed at by the property named 'name' in /chosen, or
None if the property is missing
```

dts_source(*self self*)

__repr__(*self self*)

scc_order(*self self*)

```
Returns a list of lists of Nodes where all elements of each list
depend on each other, and the Nodes in any list do not depend
on any Node in a subsequent list.  Each list defines a Strongly
Connected Component (SCC) of the graph.

For an acyclic graph each list will be a singleton.  Cycles
will be represented by lists with multiple nodes.  Cycles are
not expected to be present in devicetree graphs.
```

Public Members

dts_path

bindings_dirs

nodes

compat2enabled

class Node

```
Represents a devicetree node, augmented with information from bindings, and
with some interpretation of devicetree properties. There's a one-to-one
correspondence between devicetree nodes and Nodes.

These attributes are available on Node objects:

edt:
  The EDT instance this node is from

name:
  The name of the node

unit_addr:
  An integer with the ...@<unit-address> portion of the node name,
  translated through any 'ranges' properties on parent nodes, or None if
  the node name has no unit-address portion

description:
  The description string from the binding for the node, or None if the node
  has no binding. Leading and trailing whitespace (including newlines) is
  removed.

path:
  The devicetree path of the node

label:
  The text from the 'label' property on the node, or None if the node has
  no 'label'

labels:
  A list of all of the devicetree labels for the node, in the same order
```

(continues on next page)

(continued from previous page)

```
  as the labels appear, but with duplicates removed.
  This corresponds to the actual devicetree source labels, unlike the
  "label" attribute, which is the value of a devicetree property named
  "label".

parent:
  The Node instance for the devicetree parent of the Node, or None if the
  node is the root node

children:
  A dictionary with the Node instances for the devicetree children of the
  node, indexed by name

dep_ordinal:
  A non-negative integer value such that the value for a Node is
  less than the value for all Nodes that depend on it.

  The ordinal is defined for all Nodes including those that are not
  'enabled', and is unique among nodes in its EDT 'nodes' list.

required_by:
  A list with the nodes that directly depend on the node

depends_on:
  A list with the nodes that the node directly depends on

enabled:
  True unless the node has 'status = "disabled"'

read_only:
  True if the node has a 'read-only' property, and False otherwise

matching_compat:
  The 'compatible' string for the binding that matched the node, or None if
  the node has no binding

binding_path:
  The path to the binding file for the node, or None if the node has no
  binding

compats:
  A list of 'compatible' strings for the node, in the same order that
  they're listed in the .dts file

regs:
  A list of Register objects for the node's registers

props:
  A collections.OrderedDict that maps property names to Property objects.
  Property objects are created for all devicetree properties on the node
  that are mentioned in 'properties:' in the binding.

aliases:
  A list of aliases for the node. This is fetched from the /aliases node.

clocks:
  A list of ControllerAndData objects for the clock inputs on the
```

(continues on next page)

(continued from previous page)

```
  node, sorted by index. The list is empty if the node does not have a
  clocks property.

clock_outputs:
  A list of ControllerAndData objects for the clock outputs on the
  node, sorted by index. The list is empty if the node does not have a
  #clock-cells property.

gpio_leds:
  A list of ControllerAndData objects of the leds a gpio leds controller
  controls. The list is empty if the node is not a gpio leds controller or
  it does not have and gpio led children.

interrupts:
  A list of ControllerAndData objects for the interrupts generated by the
  node. The list is empty if the node does not generate interrupts.

pinctrls:
  A list of PinCtrl objects for the pinctrl-<index> properties on the
  node, sorted by index. The list is empty if the node does not have any
  pinctrl-<index> properties.

pinctrl_states:
  A list with the Node instances for the 'pinctrl-state' children of
  a pin controller node. The list is empty if the node does not have any
  pinctrl state children.

pinctrl_gpio_ranges:
  A list of ControllerAndData objects of the gpio ranges a pin controller
  controls. The list is empty if the node is not a pin controller or no
  'gpio-ranges' are defined by the gpio nodes.

pincfgs:
  A list of PinCfg objects for the 'pinctrl-state' node. The list is
  empty if the node is not a 'pinctrl-state' node.

pin_controller:
  The pin controller for the node. Only meaningful for nodes representing
  pinctrl states.

bus:
  If the node is a bus node (has a 'bus:' key in its binding), then this
  attribute holds the bus type, e.g. "i2c" or "spi". If the node is not a
  bus node, then this attribute is None.

on_bus:
  The bus the node appears on, e.g. "i2c" or "spi". The bus is determined
  by searching upwards for a parent node whose binding has a 'bus:' key,
  returning the value of the first 'bus:' key found. If none of the node's
  parents has a 'bus:' key, this attribute is None.

bus_node:
  Like on_bus, but contains the Node for the bus controller, or None if the
  node is not on a bus.

flash_controller:
  The flash controller for the node. Only meaningful for nodes representing
```

(continues on next page)

(continued from previous page)

```
  flash partitions.

partitions:
  A list of Partition objects of the partitions of the 'partitions' node of
  a flash. Only meaningful for nodes representing a flash - otherwise an
  empty list.
```

Public Functions

name (*self self*)

unit_addr (*self self*)

description (*self self*)

path (*self self*)

label (*self self*)

labels (*self self*)

parent (*self self*)

children (*self self*)

required_by (*self self*)

depends_on (*self self*)

enabled (*self self*)

read_only (*self self*)

aliases (*self self*)

bus (*self self*)

clocks (*self self*)

on_bus (*self self*)

flash_controller (*self self*)

__repr__ (*self self*)

Public Members

compats

matching_compat

binding_path

props

pinctrl_gpio_ranges

clock_outputs

gpio_leds

regs

partitions

name

pinctrl_states

pincfgs

pinctrls

interrupts

class Register

Represents a register on a node.

These attributes are available on Register objects:

node:
  The Node instance this register is from

name:
  The name of the register as given in the 'reg-names' property, or None if
  there is no 'reg-names' property

addr:
  The starting address of the register, in the parent address space. Any
  'ranges' properties are taken into account.

size:
  The length of the register in bytes

Public Functions

__repr__(*self self*)

class ControllerAndData

Represents an 'interrupts' or 'type: phandle-array' property
value, e.g. <&ctrl-1 4 0> in

    cs-gpios = <&ctrl-1 4 0 &ctrl-2 3 4>;

These attributes are available on ControllerAndData objects:

node:
  The Node instance the property appears on

controller:
  The Node instance for the controller (e.g. the controller the interrupt
  gets sent to for interrupts)

data:
  A dictionary that maps names from the *-cells key in the binding for the
  controller to data values, e.g. {"pin": 4, "flags": 0} for the example
  above.

  'interrupts = <1 2>' might give {"irq": 1, "level": 2}.

name:
  The name of the entry as given in
  'interrupt-names'/'gpio-names'/'pwm-names'/etc., or None if there is no
  *-names property

Public Functions

__repr__ (*self self*)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

D

E

G

I

M

N

P

R

T

Z