

---

# **cocotb Documentation**

***Release 1.1***

**PotentialVentures**

**Oct 26, 2019**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	What is cocotb? . . . . .	3
1.2	How is cocotb different? . . . . .	3
1.3	How does cocotb work? . . . . .	4
1.4	Contributors . . . . .	4
<b>2</b>	<b>Quickstart Guide</b>	<b>5</b>
2.1	Installing cocotb . . . . .	5
2.2	Running your first Example . . . . .	6
2.3	Using cocotb . . . . .	7
<b>3</b>	<b>Build options and Environment Variables</b>	<b>11</b>
3.1	Make System . . . . .	11
3.2	Environment Variables . . . . .	12
<b>4</b>	<b>Coroutines</b>	<b>15</b>
4.1	Async functions . . . . .	17
<b>5</b>	<b>Triggers</b>	<b>19</b>
5.1	Simulator Triggers . . . . .	19
5.2	Python Triggers . . . . .	20
<b>6</b>	<b>Testbench Tools</b>	<b>23</b>
6.1	Logging . . . . .	23
6.2	Buses . . . . .	24
6.3	Driving Buses . . . . .	24
6.4	Monitoring Buses . . . . .	25
6.5	Tracking testbench errors . . . . .	26
<b>7</b>	<b>Library Reference</b>	<b>27</b>
7.1	Test Results . . . . .	27
7.2	Writing and Generating tests . . . . .	28
7.3	Interacting with the Simulator . . . . .	30
7.4	Testbench Structure . . . . .	33
7.5	Utilities . . . . .	38
7.6	Simulation Object Handles . . . . .	41
7.7	Implemented Testbench Structures . . . . .	43
7.8	Miscellaneous . . . . .	48
<b>8</b>	<b>C Code Library Reference</b>	<b>49</b>
8.1	API Documentation . . . . .	49

<b>9</b>	<b>Tutorial: Endian Swapper</b>	<b>95</b>
9.1	Design	95
9.2	Testbench	95
<b>10</b>	<b>Tutorial: Ping</b>	<b>99</b>
10.1	Architecture	99
10.2	Implementation	100
10.3	Further work	102
<b>11</b>	<b>Tutorial: Driver Cosimulation</b>	<b>103</b>
11.1	Difficulties with Driver Co-simulation	103
11.2	Cocotb infrastructure	103
11.3	Implementation	104
11.4	Further Work	105
<b>12</b>	<b>More Examples</b>	<b>107</b>
12.1	Adder	107
12.2	D Flip-Flop	107
12.3	Mean	107
12.4	Mixed Language	108
12.5	AXI Lite Slave	108
12.6	Sorter	108
<b>13</b>	<b>Troubleshooting</b>	<b>111</b>
13.1	Simulation Hangs	111
13.2	Increasing Verbosity	111
13.3	Attaching a Debugger	111
<b>14</b>	<b>Simulator Support</b>	<b>113</b>
14.1	Icarus	113
14.2	Synopsys VCS	114
14.3	Aldec Riviera-PRO	114
14.4	Mentor Questa	114
14.5	Mentor ModelSim	114
14.6	Cadence Incisive, Cadence Xcelium	114
14.7	GHDL	114
<b>15</b>	<b>Roadmap</b>	<b>115</b>
<b>16</b>	<b>Release Notes</b>	<b>117</b>
16.1	cocotb 1.2	117
16.2	cocotb 1.1	118
16.3	cocotb 1.0	118
16.4	cocotb 0.4	118
16.5	cocotb 0.3	119
16.6	cocotb 0.2	119
16.7	cocotb 0.1	119
<b>17</b>	<b>Indices and tables</b>	<b>121</b>
	<b>Python Module Index</b>	<b>123</b>
	<b>Index</b>	<b>125</b>

Contents:



## INTRODUCTION

### 1.1 What is cocotb?

**cocotb** is a *CO*routine based *CO*simulation *TestBench* environment for verifying VHDL/Verilog RTL using [Python](#).

cocotb is completely free, open source (under the [BSD License](#)) and hosted on [GitHub](#).

cocotb requires a simulator to simulate the RTL. Simulators that have been tested and known to work with cocotb:

Linux Platforms

- [Icarus Verilog](#)
- [GHDL](#)
- [Aldec Riviera-PRO](#)
- [Synopsys VCS](#)
- [Cadence Incisive](#)
- [Mentor ModelSim](#) (DE and SE)

Windows Platform

- [Icarus Verilog](#)
- [Aldec Riviera-PRO](#)
- [Mentor ModelSim](#) (DE and SE)

A (possibly older) version of cocotb can be used live in a web-browser using [EDA Playground](#).

### 1.2 How is cocotb different?

cocotb encourages the same philosophy of design re-use and randomized testing as UVM, however is implemented in Python rather than SystemVerilog.

With cocotb, VHDL/Verilog/SystemVerilog are normally only used for the design itself, not the testbench.

cocotb has built-in support for integrating with the [Jenkins](#) continuous integration system.

cocotb was specifically designed to lower the overhead of creating a test.

cocotb automatically discovers tests so that no additional step is required to add a test to a regression.

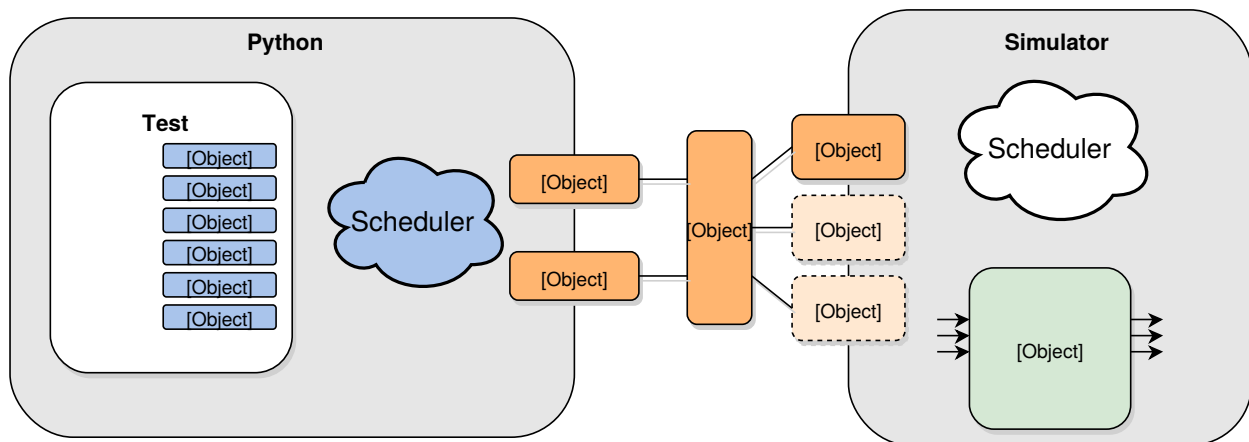
All verification is done using Python which has various advantages over using SystemVerilog or VHDL for verification:

- Writing Python is **fast** - it's a very productive language
- It's **easy** to interface to other languages from Python
- Python has a huge library of existing code to **re-use** like [packet generation](#) libraries.
- Python is **interpreted**. Tests can be edited and re-run them without having to recompile the design or exit the simulator GUI.
- Python is **popular** - far more engineers know Python than SystemVerilog or VHDL

## 1.3 How does cocotb work?

### 1.3.1 Overview

A typical cocotb testbench requires no additional RTL code. The Design Under Test (DUT) is instantiated as the toplevel in the simulator without any wrapper code. cocotb drives stimulus onto the inputs to the DUT (or further down the hierarchy) and monitors the outputs directly from Python.



A test is simply a Python function. At any given time either the simulator is advancing time or the Python code is executing. The `yield` keyword is used to indicate when to pass control of execution back to the simulator. A test can spawn multiple coroutines, allowing for independent flows of execution.

## 1.4 Contributors

cocotb was developed by [Potential Ventures](#) with the support of [Solarflare Communications Ltd](#) and contributions from Gordon McGregor and Finn Grimwood (see [contributors](#) for the full list of contributions).

We also have a list of talks and papers, libraries and examples at our Wiki page [Further Resources](#). Feel free to add links to cocotb-related content that we are still missing!



## QUICKSTART GUIDE

### 2.1 Installing cocotb

#### 2.1.1 Pre-requisites

Cocotb has the following requirements:

- Python 2.7, Python 3.5+ (recommended)
- Python-dev packages
- GCC 4.8.1+ and associated development packages
- GNU Make
- A Verilog or VHDL simulator, depending on your RTL source code

#### 2.1.2 Installation via PIP

New in version 1.2.

Cocotb can be installed by running

```
$> pip3 install cocotb
```

or

```
$> pip install cocotb
```

For user local installation follow the [pip User Guide](#).

To install the development version of cocotb:

```
$> git clone https://github.com/cocotb/cocotb
$> pip install -e ./cocotb
```

#### 2.1.3 Native Linux Installation

The following instructions will allow building of the cocotb libraries for use with a 64-bit native simulator.

If a 32-bit simulator is being used then additional steps are needed, please see [our Wiki](#).

Debian/Ubuntu-based

```
$> sudo apt-get install git make gcc g++ swig python-dev
```

Red Hat-based

```
$> sudo yum install gcc gcc-c++ libstdc++-devel swig python-devel
```

## 2.1.4 Windows Installation

Download the MinGW installer from <https://osdn.net/projects/mingw/releases/>.

Run the GUI installer and specify a directory you would like the environment installed in. The installer will retrieve a list of possible packages, when this is done press “Continue”. The MinGW Installation Manager is then launched.

The following packages need selecting by checking the tick box and selecting “Mark for installation”

```
Basic Installation
-- mingw-developer-tools
-- mingw32-base
-- mingw32-gcc-g++
-- msys-base
```

From the Installation menu then select “Apply Changes”, in the next dialog select “Apply”.

When installed a shell can be opened using the `msys.bat` file located under the `<install_dir>/msys/1.0/`

Python can be downloaded from <https://www.python.org/downloads/windows/>. Run the installer and download to your chosen location.

It is beneficial to add the path to Python to the Windows system `PATH` variable so it can be used easily from inside Msys.

Once inside the Msys shell commands as given here will work as expected.

## 2.1.5 macOS Packages

You need a few packages installed to get cocotb running on macOS. Installing a package manager really helps things out here.

[Brew](#) seems to be the most popular, so we’ll assume you have that installed.

```
$> brew install python icarus-verilog gtkwave
```

## 2.2 Running your first Example

Assuming you have installed the prerequisites as above, the following lines are all you need to run a first simulation with cocotb:

```
$> git clone https://github.com/cocotb/cocotb
$> cd cocotb/examples/endian_swapper/tests
$> make
```

Selecting a different simulator is as easy as:

```
$> make SIM=vcs
```

## 2.2.1 Running the same example as VHDL

The `endian_swapper` example includes both a VHDL and a Verilog RTL implementation. The cocotb testbench can execute against either implementation using VPI for Verilog and VHPI/FLI for VHDL. To run the test suite against the VHDL implementation use the following command (a VHPI or FLI capable simulator must be used):

```
$> make SIM=ghdl TOPLEVEL_LANG=vhdl
```

## 2.3 Using cocotb

A typical cocotb testbench requires no additional HDL code (though nothing prevents you from adding testbench helper code). The Design Under Test (DUT) is instantiated as the toplevel in the simulator without any wrapper code. Cocotb drives stimulus onto the inputs to the DUT and monitors the outputs directly from Python.

### 2.3.1 Creating a Makefile

To create a cocotb test we typically have to create a Makefile. Cocotb provides rules which make it easy to get started. We simply inform cocotb of the source files we need compiling, the toplevel entity to instantiate and the Python test script to load.

```
VERILOG_SOURCES = $(PWD)/submodule.sv $(PWD)/my_design.sv
# TOPLEVEL is the name of the toplevel module in your Verilog or VHDL file:
TOPLEVEL=my_design
# MODULE is the name of the Python test file:
MODULE=test_my_design

include $(shell cocotb-config --makefiles)/Makefile.inc
include $(shell cocotb-config --makefiles)/Makefile.sim
```

We would then create a file called `test_my_design.py` containing our tests.

### 2.3.2 Creating a test

The test is written in Python. Cocotb wraps your top level with the handle you pass it. In this documentation, and most of the examples in the project, that handle is `dut`, but you can pass your own preferred name in instead. The handle is used in all Python files referencing your RTL project. Assuming we have a toplevel port called `clk` we could create a test file containing the following:

```
import cocotb
from cocotb.triggers import Timer

@cocotb.test()
def my_first_test(dut):
    """Try accessing the design."""

    dut._log.info("Running test!")
    for cycle in range(10):
        dut.clk = 0
```

(continues on next page)

(continued from previous page)

```
yield Timer(1, units='ns')
dut.clk = 1
yield Timer(1, units='ns')
dut._log.info("Running test!")
```

This will drive a square wave clock onto the `clk` port of the toplevel.

### 2.3.3 Accessing the design

When cocotb initializes it finds the top-level instantiation in the simulator and creates a handle called `dut`. Top-level signals can be accessed using the “dot” notation used for accessing object attributes in Python. The same mechanism can be used to access signals inside the design.

```
# Get a reference to the "clk" signal on the top-level
clk = dut.clk

# Get a reference to a register "count"
# in a sub-block "inst_sub_block"
count = dut.inst_sub_block.count
```

### 2.3.4 Assigning values to signals

Values can be assigned to signals using either the `value` property of a handle object or using direct assignment while traversing the hierarchy.

```
# Get a reference to the "clk" signal and assign a value
clk = dut.clk
clk.value = 1

# Direct assignment through the hierarchy
dut.input_signal <= 12

# Assign a value to a memory deep in the hierarchy
dut.sub_block.memory.array[4] <= 2
```

The syntax `sig <= new_value` is a short form of `sig.value = new_value`. It not only resembles HDL-syntax, but also has the same semantics: writes are not applied immediately, but delayed until the next write cycle. Use `sig.setimmediatevalue(new_val)` to set a new value immediately (see `setimmediatevalue()`).

### 2.3.5 Reading values from signals

Accessing the `value` property of a handle object will return a `BinaryValue` object. Any unresolved bits are preserved and can be accessed using the `binstr` attribute, or a resolved integer value can be accessed using the `integer` attribute.

```
>>> # Read a value back from the DUT
>>> count = dut.counter.value
>>>
>>> print(count.binstr)
1X1010
>>> # Resolve the value to an integer (X or Z treated as 0)
>>> print(count.integer)
```

(continues on next page)

(continued from previous page)

```
42
>>> # Show number of bits in a value
>>> print(count.n_bits)
6
```

We can also cast the signal handle directly to an integer:

```
>>> print(int(dut.counter))
42
```

### 2.3.6 Parallel and sequential execution of coroutines

```
@cocotb.coroutine
def reset_dut(reset_n, duration):
    reset_n <= 0
    yield Timer(duration, units='ns')
    reset_n <= 1
    reset_n._log.debug("Reset complete")

@cocotb.test()
def parallel_example(dut):
    reset_n = dut.reset

    # This will call reset_dut sequentially
    # Execution will block until reset_dut has completed
    yield reset_dut(reset_n, 500)
    dut._log.debug("After reset")

    # Call reset_dut in parallel with this coroutine
    reset_thread = cocotb.fork(reset_dut(reset_n, 500))

    yield Timer(250, units='ns')
    dut._log.debug("During reset (reset_n = %s)" % reset_n.value)

    # Wait for the other thread to complete
    yield reset_thread.join()
    dut._log.debug("After reset")
```



## BUILD OPTIONS AND ENVIRONMENT VARIABLES

### 3.1 Make System

Makefiles are provided for a variety of simulators in `cocotb/share/makefiles/simulators`. The common Makefile `cocotb/share/makefiles/Makefile.sim` includes the appropriate simulator Makefile based on the contents of the `SIM` variable.

#### 3.1.1 Make Targets

Makefiles define two targets, `regression` and `sim`, the default target is `sim`.

Both rules create a results file with the name taken from `COCOTB_RESULTS_FILE`, defaulting to `results.xml`. This file is a JUnit-compatible output file suitable for use with [Jenkins](#). The `sim` targets unconditionally re-runs the simulator whereas the `regression` target only re-builds if any dependencies have changed.

#### 3.1.2 Make Phases

Typically the makefiles provided with Cocotb for various simulators use a separate `compile` and `run` target. This allows for a rapid re-running of a simulator if none of the RTL source files have changed and therefore the simulator does not need to recompile the RTL.

#### 3.1.3 Make Variables

**GUI** Set this to 1 to enable the GUI mode in the simulator (if supported).

**SIM** Selects which simulator Makefile to use. Attempts to include a simulator specific makefile from `cocotb/share/makefiles/makefile.$(SIM)`

**WAVES** Set this to 1 to enable wave traces dump for the Aldec Riviera-PRO and Mentor Graphics Questa simulators. To get wave traces in Icarus Verilog see [Simulator Support](#).

**VERILOG\_SOURCES** A list of the Verilog source files to include.

**VHDL\_SOURCES** A list of the VHDL source files to include.

**VHDL\_SOURCES\_lib** A list of the VHDL source files to include in the VHDL library *lib* (currently GHDL only).

**COMPILE\_ARGS** Any arguments or flags to pass to the compile stage of the simulation.

**SIM\_ARGS** Any arguments or flags to pass to the execution of the compiled simulation.

**EXTRA\_ARGS** Passed to both the compile and execute phases of simulators with two rules, or passed to the single compile and run command for simulators which don't have a distinct compilation stage.

**CUSTOM\_COMPILE\_DEPS** Use to add additional dependencies to the compilation target; useful for defining additional rules to run pre-compilation or if the compilation phase depends on files other than the RTL sources listed in `VERILOG_SOURCES` or `VHDL_SOURCES`.

**CUSTOM\_SIM\_DEPS** Use to add additional dependencies to the simulation target.

**COCOTB\_NVC\_TRACE** Set this to 1 to enable display of VHPI traces when using the NVC VHDL simulator.

**SIM\_BUILD** Use to define a scratch directory for use by the simulator. The path is relative to the Makefile location. If not provided, the default scratch directory is `sim_build`.

## 3.2 Environment Variables

### TOPLEVEL

Used to indicate the instance in the hierarchy to use as the DUT. If this isn't defined then the first root instance is used.

### RANDOM\_SEED

Seed the Python random module to recreate a previous test stimulus. At the beginning of every test a message is displayed with the seed used for that execution:

```
INFO      cocotb.gpi                               __init__.py:89   in _
↪initialise_testbench          Seeding Python random module with 1377424946
```

To recreate the same stimuli use the following:

```
make RANDOM_SEED=1377424946
```

### COCOTB\_ANSI\_OUTPUT

Use this to override the default behavior of annotating cocotb output with ANSI color codes if the output is a terminal (`isatty()`).

`COCOTB_ANSI_OUTPUT=1` forces output to be ANSI regardless of the type of `stdout`

`COCOTB_ANSI_OUTPUT=0` suppresses the ANSI output in the log messages

### COCOTB\_REDUCED\_LOG\_FMT

If defined, log lines displayed in terminal will be shorter. It will print only time, message type (INFO, WARNING, ERROR) and log message.

### MODULE

The name of the module(s) to search for test functions. Multiple modules can be specified using a comma-separated list.

### TESTCASE

The name of the test function(s) to run. If this variable is not defined Cocotb discovers and executes all functions decorated with the `cocotb.test` decorator in the supplied modules.

Multiple functions can be specified in a comma-separated list.

### COCOTB\_RESULTS\_FILE

The file name where XML tests results are stored. If not provided, the default is `results.xml`.

New in version 1.3.



### 3.2.1 Additional Environment Variables

#### **COCOTB\_ATTACH**

In order to give yourself time to attach a debugger to the simulator process before it starts to run, you can set the environment variable `COCOTB_ATTACH` to a pause time value in seconds. If set, Cocotb will print the process ID (PID) to attach to and wait the specified time before actually letting the simulator run.

#### **COCOTB\_ENABLE\_PROFILING**

Enable performance analysis of the Python portion of Cocotb. When set, a file `test_profile.pstat` will be written which contains statistics about the cumulative time spent in the functions.

From this, a callgraph diagram can be generated with `gprof2dot` and `graphviz`. See the `profile` Make target in the `endian_swapper` example on how to set this up.

#### **COCOTB\_HOOKS**

A comma-separated list of modules that should be executed before the first test. You can also use the `cocotb.hook` decorator to mark a function to be run before test code.

#### **COCOTB\_LOG\_LEVEL**

Default logging level to use. This is set to `INFO` unless overridden.

#### **COCOTB\_RESOLVE\_X**

Defines how to resolve bits with a value of X, Z, U or W when being converted to integer. Valid settings are:

**VALUE\_ERROR** raise a `ValueError` exception

**ZEROS** resolve to 0

**ONES** resolve to 1

**RANDOM** randomly resolve to a 0 or a 1

Set to `VALUE_ERROR` by default.

#### **COCOTB\_SCHEDULER\_DEBUG**

Enable additional log output of the coroutine scheduler.

#### **COVERAGE**

Enable to report python coverage data. For some simulators, this will also report HDL coverage.

This needs the `coverage` python module

#### **MEMCHECK**

HTTP port to use for debugging Python's memory usage. When set to e.g. 8088, data will be presented at <http://localhost:8088>.

This needs the `cherrypy` and `dowser` Python modules installed.

#### **COCOTB\_PY\_DIR**

Path to the directory containing the cocotb Python package in the `cocotb` subdirectory.

#### **COCOTB\_SHARE\_DIR**

Path to the directory containing the cocotb Makefiles and simulator libraries in the subdirectories `lib`, `include`, and `makefiles`.

#### **VERSION**

The version of the Cocotb installation. You probably don't want to modify this.



## COROUTINES

Testbenches built using cocotb use coroutines. While the coroutine is executing the simulation is paused. The coroutine uses the `yield` keyword to pass control of execution back to the simulator and simulation time can advance again.

Typically coroutines `yield` a *Trigger* object which indicates to the simulator some event which will cause the coroutine to be woken when it occurs. For example:

```
@cocotb.coroutine
def wait_10ns():
    cocotb.log.info("About to wait for 10 ns")
    yield Timer(10, units='ns')
    cocotb.log.info("Simulation time has advanced by 10 ns")
```

Coroutines may also yield other coroutines:

```
@cocotb.coroutine
def wait_100ns():
    for i in range(10):
        yield wait_10ns()
```

Coroutines can return a value, so that they can be used by other coroutines. Before Python 3.3, this requires a *ReturnValue* to be raised.

```
@cocotb.coroutine
def get_signal(clk, signal):
    yield RisingEdge(clk)
    raise ReturnValue(signal.value)

@cocotb.coroutine
def get_signal_python_33(clk, signal):
    # newer versions of Python can use return normally
    yield RisingEdge(clk)
    return signal.value

@cocotb.coroutine
def check_signal_changes(dut):
    first = yield get_signal(dut.clk, dut.signal)
    second = yield get_signal(dut.clk, dut.signal)
    if first == second:
        raise TestFailure("Signal did not change")
```

Coroutines may also yield a list of triggers and coroutines to indicate that execution should resume if *any* of them fires:

```
@cocotb.coroutine
def packet_with_timeout(monitor, timeout):
```

(continues on next page)

(continued from previous page)

```

"""Wait for a packet but time out if nothing arrives"""
yield [Timer(timeout, units='ns'), RisingEdge(dut.ready)]

```

The trigger that caused execution to resume is passed back to the coroutine, allowing them to distinguish which trigger fired:

```

@cocotb.coroutine
def packet_with_timeout(monitor, timeout):
    """Wait for a packet but time out if nothing arrives"""
    tout_trigger = Timer(timeout, units='ns')
    result = yield [tout_trigger, RisingEdge(dut.ready)]
    if result is tout_trigger:
        raise TestFailure("Timed out waiting for packet")

```

Coroutines can be forked for parallel operation within a function of that code and the forked code.

```

@cocotb.test()
def test_act_during_reset(dut):
    """While reset is active, toggle signals"""
    tb = uart_tb(dut)
    # "Clock" is a built in class for toggling a clock signal
    cocotb.fork(Clock(dut.clk, 1, units='ns').start())
    # reset_dut is a function -
    # part of the user-generated "uart_tb" class
    cocotb.fork(tb.reset_dut(dut.rstn, 20))

    yield Timer(10, units='ns')
    print("Reset is still active: %d" % dut.rstn)
    yield Timer(15, units='ns')
    print("Reset has gone inactive: %d" % dut.rstn)

```

Coroutines can be joined to end parallel operation within a function.

```

@cocotb.test()
def test_count_edge_cycles(dut, period=1, clocks=6):
    cocotb.fork(Clock(dut.clk, period, units='ns').start())
    yield RisingEdge(dut.clk)

    timer = Timer(period + 10)
    task = cocotb.fork(count_edges_cycles(dut.clk, clocks))
    count = 0
    expect = clocks - 1

    while True:
        result = yield [timer, task.join()]
        if count > expect:
            raise TestFailure("Task didn't complete in expected time")
        if result is timer:
            dut._log.info("Count %d: Task still running" % count)
            count += 1
        else:
            break

```

Coroutines can be killed before they complete, forcing their completion before they'd naturally end.

```

@cocotb.test()
def test_different_clocks(dut):
    clk_1mhz = Clock(dut.clk, 1.0, units='us')
    clk_250mhz = Clock(dut.clk, 4.0, units='ns')

    clk_gen = cocotb.fork(clk_1mhz.start())
    start_time_ns = get_sim_time(units='ns')
    yield Timer(1, units='ns')
    yield RisingEdge(dut.clk)
    edge_time_ns = get_sim_time(units='ns')
    # NOTE: isclose is a Python 3.5+ feature
    if not isclose(edge_time_ns, start_time_ns + 1000.0):
        raise TestFailure("Expected a period of 1 us")

    clk_gen.kill()

    clk_gen = cocotb.fork(clk_250mhz.start())
    start_time_ns = get_sim_time(units='ns')
    yield Timer(1, units='ns')
    yield RisingEdge(dut.clk)
    edge_time_ns = get_sim_time(units='ns')
    # NOTE: isclose is a Python 3.5+ feature
    if not isclose(edge_time_ns, start_time_ns + 4.0):
        raise TestFailure("Expected a period of 4 ns")

```

## 4.1 Async functions

Python 3.5 introduces `async` functions, which provide an alternative syntax. For example:

```

@cocotb.coroutine
async def wait_10ns():
    cocotb.log.info("About to wait for 10 ns")
    await Timer(10, units='ns')
    cocotb.log.info("Simulation time has advanced by 10 ns")

```

To wait on a trigger or a nested coroutine, these use `await` instead of `yield`. Provided they are decorated with `@cocotb.coroutine`, `async def` functions using `await` and regular functions using `yield` can be used interchangeably - the appropriate keyword to use is determined by which type of function it appears in, not by the sub-coroutine being called.

---

**Note:** It is not legal to `await` a list of triggers as can be done in `yield`-based coroutines with `yield [trig1, trig2]`. Use `await First(trig1, trig2)` instead.

---

### 4.1.1 Async generators

In Python 3.6, a `yield` statement within an `async` function has a new meaning (rather than being a `SyntaxError`) which matches the typical meaning of `yield` within regular Python code. It can be used to create a special type of generator function that can be iterated with `async for`:

```

async def ten_samples_of(clk, signal):
    for i in range(10):

```

(continues on next page)

(continued from previous page)

```
    await RisingEdge(clk)
    yield signal.value # this means "send back to the for loop"

@cocotb.test()
async def test_samples_are_even(dut):
    async for sample in ten_samples_of(dut.clk, dut.signal):
        assert sample % 2 == 0
```

More details on this type of generator can be found in [PEP 525](#).

## TRIGGERS

Triggers are used to indicate when the cocotb scheduler should resume coroutine execution. To use a trigger, a coroutine should `await` or `yield` it. This will cause execution of the current coroutine to pause. When the trigger fires, execution of the paused coroutine will resume:

```
@cocotb.coroutine
def coro():
    print("Some time before the edge")
    yield RisingEdge(clk)
    print("Immediately after the edge")
```

Or using the syntax in Python 3.5 onwards:

```
@cocotb.coroutine
async def coro():
    print("Some time before the edge")
    await RisingEdge(clk)
    print("Immediately after the edge")
```

## 5.1 Simulator Triggers

### 5.1.1 Signals

**class** cocotb.triggers.**Edge**(*signal*)  
Fires on any value change of *signal*.

**class** cocotb.triggers.**RisingEdge**(*signal*)  
Fires on the rising edge of *signal*, on a transition from 0 to 1.

**class** cocotb.triggers.**FallingEdge**(*signal*)  
Fires on the falling edge of *signal*, on a transition from 1 to 0.

**class** cocotb.triggers.**ClockCycles**(*signal*, *num\_cycles*, *rising=True*)  
Fires after *num\_cycles* transitions of *signal* from 0 to 1.

**Parameters** **rising** – If true, the default, count rising edges. Otherwise, count falling edges

### 5.1.2 Timing

**class** cocotb.triggers.**Timer**(*time\_ps*, *units=None*)  
Fires after the specified simulation time period has elapsed.

**class** cocotb.triggers.ReadOnly

Fires when the current simulation timestep moves to the read-only phase.

The read-only phase is entered when the current timestep no longer has any further delta steps. This will be a point where all the signal values are stable as there are no more RTL events scheduled for the timestep. The simulator will not allow scheduling of more events in this timestep. Useful for monitors which need to wait for all processes to execute (both RTL and cocotb) to ensure sampled signal values are final.

**class** cocotb.triggers.ReadWrite

Fires when the read-write portion of the sim cycles is reached.

**class** cocotb.triggers.NextTimeStep

Fires when the next time step is started.

## 5.2 Python Triggers

**class** cocotb.triggers.Combine(\*triggers)

Fires when all of *triggers* have fired.

Like most triggers, this simply returns itself.

**class** cocotb.triggers.First(\*triggers)

Fires when the first trigger in *triggers* fires.

Returns the result of the trigger that fired.

As a shorthand, `t = yield [a, b]` can be used instead of `t = yield First(a, b)`. Note that this shorthand is not available when using `await`.

---

**Note:** The event loop is single threaded, so while events may be simultaneous in simulation time, they can never be simultaneous in real time. For this reason, the value of `t_ret` is `t1` in the following example is implementation-defined, and will vary by simulator:

```
t1 = Timer(10, units='ps')
t2 = Timer(10, units='ps')
t_ret = yield First(t1, t2)
```

**class** cocotb.triggers.Join(coroutine)

Fires when a `fork()`ed coroutine completes

The result of blocking on the trigger can be used to get the coroutine result:

```
@cocotb.coroutine()
def coro_inner():
    yield Timer(1)
    raise ReturnValue("Hello world")

task = cocotb.fork(coro_inner())
result = yield Join(task)
assert result == "Hello world"
```

Or using the syntax in Python 3.5 onwards:

```
@cocotb.coroutine()
async def coro_inner():
    await Timer(1)
```

(continues on next page)



(continued from previous page)

```

    return "Hello world"

task = cocotb.fork(coro_inner())
result = await Join(task)
assert result == "Hello world"

```

If the coroutine threw an exception, the `await` or `yield` will re-raise it.

#### property `retval`

The return value of the joined coroutine.

**Note:** Typically there is no need to use this attribute - the following code samples are equivalent:

```

forked = cocotb.fork(mycoro())
j = Join(forked)
yield j
result = j.retval

```

```

forked = cocotb.fork(mycoro())
result = yield Join(forked)

```

## 5.2.1 Synchronization

These are not `Triggers` themselves, but contain methods that can be used as triggers. These are used to synchronize coroutines with each other.

**class** `cocotb.triggers.Event` (*name=""*)

Event to permit synchronization between two coroutines.

Yielding `wait()` from one coroutine will block the coroutine until `set()` is called somewhere else.

**set** (*data=None*)

Wake up all coroutines blocked on this event.

**wait** ()

Get a trigger which fires when another coroutine sets the event.

If the event has already been set, the trigger will fire immediately.

To reset the event (and enable the use of `wait` again), `clear()` should be called.

**clear** ()

Clear this event that has fired.

Subsequent calls to `wait()` will block until `set()` is called again.

**class** `cocotb.triggers.Lock` (*name=""*)

Lock primitive (not re-entrant).

This should be used as:

```

yield lock.acquire()
try:
    # do some stuff
finally:
    lock.release()

```

**locked = None**

True if the lock is held

**acquire()**

Produce a trigger which fires when the lock is acquired.

**release()**

Release the lock.

## TESTBENCH TOOLS

### 6.1 Logging

Cocotb extends the Python logging library. Each DUT, monitor, driver, and scoreboard (as well as any other function using the coroutine decorator) implements its own logging object, and each can be set to its own logging level. Within a DUT, each hierarchical object can also have individual logging levels set.

When logging HDL objects, beware that `_log` is the preferred way to use logging. This helps minimize the change of name collisions with an HDL log component with the Python logging functionality.

Log printing levels can also be set on a per-object basis.

```
class EndianSwapperTB(object):

    def __init__(self, dut, debug=False):
        self.dut = dut
        self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)
        self.stream_in_recovered = AvalonSTMonitor(dut, "stream_in", dut.clk,
                                                    callback=self.model)

        # Set verbosity on our various interfaces
        level = logging.DEBUG if debug else logging.WARNING
        self.stream_in.log.setLevel(level)
        self.stream_in_recovered.log.setLevel(level)
        self.dut.reset_n._log.setLevel(logging.DEBUG)
```

And when the logging is actually called

```
class AvalonSTPkts(BusMonitor):
    ...
    @coroutine
    def _monitor_recv(self):
        ...
        self.log.info("Received a packet of %d bytes" % len(pkt))

class Scoreboard(object):
    ...
    def add_interface(self):
        ...
        self.log.info("Created with reorder_depth %d" % reorder_depth)

class EndianSwapTB(object):
    ...
    @cocotb.coroutine
```

(continues on next page)

(continued from previous page)

```
def reset():
    self.dut._log.debug("Resetting DUT")
```

will display as something like

```
0.00ns INFO          cocotb.scoreboard.endian_swapper_sv          scoreboard.
↳py:177 in add_interface          Created with reorder_depth 0
0.00ns DEBUG          cocotb.endian_swapper_sv          ..endian_swapper.
↳py:106 in reset          Resetting DUT
16000000000000.00ns INFO          cocotb.endian_swapper_sv.stream_out          avalon.
↳py:151 in _monitor_recv          Received a packet of 125 bytes
```

## 6.2 Buses

Buses are simply defined as collection of signals. The `Bus` class will automatically bundle any group of signals together that are named similar to `dut.<bus_name><separator><signal_name>`. For instance,

```
dut.stream_in_valid
dut.stream_in_data
```

have a bus name of `stream_in`, a separator of `_`, and signal names of `valid` and `data`. A list of signal names, or a dictionary mapping attribute names to signal names is also passed into the `Bus` class. Buses can have values driven onto them, be captured (returning a dictionary), or sampled and stored into a similar object.

```
stream_in_bus = Bus(dut, "stream_in", ["valid", "data"]) # '_' is the default_
↳separator
```

## 6.3 Driving Buses

Examples and specific bus implementation bus drivers (AMBA, Avalon, XGMII, and others) exist in the `Driver` class enabling a test to append transactions to perform the serialization of transactions onto a physical interface. Here is an example using the Avalon bus driver in the `endian_swapper` example:

```
class EndianSwapperTB(object):

    def __init__(self, dut, debug=False):
        self.dut = dut
        self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)

    def run_test(dut, data_in=None, config_coroutine=None, idle_inserter=None,
                backpressure_inserter=None):

        cocotb.fork(Clock(dut.clk, 5000).start())
        tb = EndianSwapperTB(dut)

        yield tb.reset()
        dut.stream_out_ready <= 1

        if idle_inserter is not None:
            tb.stream_in.set_valid_generator(idle_inserter())
```

(continues on next page)

(continued from previous page)

```
# Send in the packets
for transaction in data_in():
    yield tb.stream_in.send(transaction)
```

## 6.4 Monitoring Buses

For our testbenches to actually be useful, we have to monitor some of these buses, and not just drive them. That's where the *Monitor* class comes in, with pre-built monitors for Avalon and XGMII buses. The *Monitor* class is a base class which you are expected to derive for your particular purpose. You must create a *\_monitor\_recv()* function which is responsible for determining 1) at what points in simulation to call the *\_recv()* function, and 2) what transaction values to pass to be stored in the monitors receiving queue. Monitors are good for both outputs of the DUT for verification, and for the inputs of the DUT, to drive a test model of the DUT to be compared to the actual DUT. For this purpose, input monitors will often have a callback function passed that is a model. This model will often generate expected transactions, which are then compared using the *Scoreboard* class.

```
# =====
class BitMonitor(Monitor):
    """Observes single input or output of DUT."""
    def __init__(self, name, signal, clock, callback=None, event=None):
        self.name = name
        self.signal = signal
        self.clock = clock
        Monitor.__init__(self, callback, event)

    @coroutine
    def _monitor_recv(self):
        clkedge = RisingEdge(self.clock)

        while True:
            # Capture signal at rising edge of clock
            yield clkedge
            vec = self.signal.value
            self._recv(vec)

# =====
def input_gen():
    """Generator for input data applied by BitDriver"""
    while True:
        yield random.randint(1,5), random.randint(1,5)

# =====
class DFF_TB(object):
    def __init__(self, dut, init_val):

        self.dut = dut

        # Create input driver and output monitor
        self.input_drv = BitDriver(dut.d, dut.c, input_gen())
        self.output_mon = BitMonitor("output", dut.q, dut.c)

        # Create a scoreboard on the outputs
        self.expected_output = [ init_val ]
```

(continues on next page)

(continued from previous page)

```
# Reconstruct the input transactions from the pins
# and send them to our 'model'
self.input_mon = BitMonitor("input", dut.d, dut.c, callback=self.model)

def model(self, transaction):
    """Model the DUT based on the input transaction."""
    # Do not append an output transaction for the last clock cycle of the
    # simulation, that is, after stop() has been called.
    if not self.stopped:
        self.expected_output.append(transaction)
```

## 6.5 Tracking testbench errors

The *Scoreboard* class is used to compare the actual outputs to expected outputs. Monitors are added to the scoreboard for the actual outputs, and the expected outputs can be either a simple list, or a function that provides a transaction. Here is some code from the *dff* example, similar to above with the scoreboard added.

```
class DFF_TB(object):
    def __init__(self, dut, init_val):
        self.dut = dut

        # Create input driver and output monitor
        self.input_drv = BitDriver(dut.d, dut.c, input_gen())
        self.output_mon = BitMonitor("output", dut.q, dut.c)

        # Create a scoreboard on the outputs
        self.expected_output = [ init_val ]
        self.scoreboard = Scoreboard(dut)
        self.scoreboard.add_interface(self.output_mon, self.expected_output)

        # Reconstruct the input transactions from the pins
        # and send them to our 'model'
        self.input_mon = BitMonitor("input", dut.d, dut.c, callback=self.model)
```

## LIBRARY REFERENCE

### 7.1 Test Results

The exceptions in this module can be raised at any point by any code and will terminate the test.

`cocotb.result.raise_error(obj, msg)`

Creates a `TestError` exception and raises it after printing a traceback.

Deprecated since version 1.3: Use `raise TestError(msg)` instead of this function. A stacktrace will be printed by cocotb automatically if the exception is unhandled.

#### Parameters

- **obj** – Object with a log method.
- **msg** (*str*) – The log message.

`cocotb.result.create_error(obj, msg)`

Like `raise_error()`, but return the exception rather than raise it, simply to avoid too many levels of nested `try/except` blocks.

Deprecated since version 1.3: Use `TestError(msg)` directly instead of this function.

#### Parameters

- **obj** – Object with a log method.
- **msg** (*str*) – The log message.

**exception** `cocotb.result.ReturnValue(retval)`

Helper exception needed for Python versions prior to 3.3.

**exception** `cocotb.result.TestComplete(*args, **kwargs)`

Exception showing that test was completed. Sub-exceptions detail the exit status.

**exception** `cocotb.result.ExternalException(exception)`

Exception thrown by external functions.

**exception** `cocotb.result.TestError(*args, **kwargs)`

Exception showing that test was completed with severity Error.

**exception** `cocotb.result.TestFailure(*args, **kwargs)`

Exception showing that test was completed with severity Failure.

**exception** `cocotb.result.TestSuccess(*args, **kwargs)`

Exception showing that test was completed successfully.

**exception** `cocotb.result.SimFailure(*args, **kwargs)`

Exception showing that simulator exited unsuccessfully.

## 7.2 Writing and Generating tests

**class** `cocotb.test` (*f*, *timeout=None*, *expect\_fail=False*, *expect\_error=False*, *skip=False*, *stage=None*)

Decorator to mark a function as a test.

All tests are coroutines. The test decorator provides some common reporting etc., a test timeout and allows us to mark tests as expected failures.

Used as `@cocotb.test(...)`.

### Parameters

- **timeout** (*int*, *optional*) – value representing simulation timeout (not implemented).
- **expect\_fail** (*bool*, *optional*) – Don't mark the result as a failure if the test fails.
- **expect\_error** (*bool*, *optional*) – Don't mark the result as an error if an error is raised. This is for cocotb internal regression use when a simulator error is expected.
- **skip** (*bool*, *optional*) – Don't execute this test as part of the regression.
- **stage** (*int*, *optional*) – Order tests logically into stages, where multiple tests can share a stage.

**class** `cocotb.coroutine` (*func*)

Decorator class that allows us to provide common coroutine mechanisms:

`log` methods will log to `cocotb.coroutine.name`.

`join()` method returns an event which will fire when the coroutine exits.

Used as `@cocotb.coroutine`.

**class** `cocotb.external` (*func*)

Decorator to apply to an external function to enable calling from cocotb. This currently creates a new execution context for each function that is called. Scope for this to be streamlined to a queue in future.

**class** `cocotb.function` (*func*)

Decorator class that allows a function to block.

This allows a function to internally block while externally appear to yield.

**class** `cocotb.hook` (*f*)

Decorator to mark a function as a hook for cocotb.

Used as `@cocotb.hook()`.

All hooks are run at the beginning of a cocotb test suite, prior to any test code being run.

**class** `cocotb.regression.TestFactory` (*test\_function*, *\*args*, *\*\*kwargs*)

Used to automatically generate tests.

Assuming we have a common test function that will run a test. This test function will take keyword arguments (for example generators for each of the input interfaces) and generate tests that call the supplied function.

This Factory allows us to generate sets of tests based on the different permutations of the possible arguments to the test function.

For example if we have a module that takes backpressure and idles and have some packet generation routines `gen_a` and `gen_b`:

```
>>> tf = TestFactory(run_test)
>>> tf.add_option('data_in', [gen_a, gen_b])
>>> tf.add_option('backpressure', [None, random_backpressure])
```

(continues on next page)



(continued from previous page)

```
>>> tf.add_option('idles', [None, random_idles])
>>> tf.generate_tests()
```

We would get the following tests:

- `gen_a` with no backpressure and no idles
- `gen_a` with no backpressure and `random_idles`
- `gen_a` with `random_backpressure` and no idles
- `gen_a` with `random_backpressure` and `random_idles`
- `gen_b` with no backpressure and no idles
- `gen_b` with no backpressure and `random_idles`
- `gen_b` with `random_backpressure` and no idles
- `gen_b` with `random_backpressure` and `random_idles`

The tests are appended to the calling module for auto-discovery.

Tests are simply named `test_function_N`. The docstring for the test (hence the test description) includes the name and description of each generator.

#### Parameters

- **test\_function** (*function*) – the function that executes a test. Must take *dut* as the first argument.
- **\*args** – Remaining arguments are passed directly to the test function. Note that these arguments are not varied. An argument that varies with each test must be a keyword argument to the test function.
- **\*kwargs** – Remaining keyword arguments are passed directly to the test function. Note that these arguments are not varied. An argument that varies with each test must be a keyword argument to the test function.

**add\_option** (*name*, *optionlist*)

Add a named option to the test.

#### Parameters

- **name** (*str*) – Name of the option. Passed to test as a keyword argument.
- **optionlist** (*list*) – A list of possible options for this test knob.

**generate\_tests** (*prefix*=",", *postfix*=")

Generates exhaustive set of tests using the cartesian product of the possible keyword arguments.

The generated tests are appended to the namespace of the calling module.

#### Parameters

- **prefix** (*str*) – Text string to append to start of `test_function` name when naming generated test cases. This allows reuse of a single `test_function` with multiple *TestFactories* without name clashes.
- **postfix** (*str*) – Text string to append to end of `test_function` name when naming generated test cases. This allows reuse of a single `test_function` with multiple *TestFactories* without name clashes.

## 7.3 Interacting with the Simulator

**class** cocotb.binary.**BinaryRepresentation**

**UNSIGNED** = 0

Unsigned format

**SIGNED\_MAGNITUDE** = 1

Sign and magnitude format

**TWOS\_COMPLEMENT** = 2

Two's complement format

**class** cocotb.binary.**BinaryValue** (*value=None, n\_bits=None, bigEndian=True, binaryRepresentation=0, bits=None*)

Representation of values in binary format.

The underlying value can be set or accessed using these aliasing attributes:

- `BinaryValue.integer` is an integer
- `BinaryValue.signed_integer` is a signed integer
- `BinaryValue.binstr` is a string of 01xXzZ
- `BinaryValue.buff` is a binary buffer of bytes
- `BinaryValue.value` is an integer **deprecated**

For example:

```
>>> vec = BinaryValue()
>>> vec.integer = 42
>>> print (vec.binstr)
101010
>>> print (repr (vec.buff))
'\x0a'
```

### Parameters

- **value** (*str or int or long, optional*) – Value to assign to the bus.
- **n\_bits** (*int, optional*) – Number of bits to use for the underlying binary representation.
- **bigEndian** (*bool, optional*) – Interpret the binary as big-endian when converting to/from a string buffer.
- **binaryRepresentation** (`BinaryRepresentation`) – The representation of the binary value (one of `UNSIGNED`, `SIGNED_MAGNITUDE`, `TWOS_COMPLEMENT`). Defaults to unsigned representation.
- **bits** (*int, optional*) – Deprecated: Compatibility wrapper for `n_bits`.

**assign** (*value*)

Decides how best to assign the value to the vector.

We possibly try to be a bit too clever here by first of all trying to assign the raw string as a `BinaryValue.binstr`, however if the string contains any characters that aren't 0, 1, X or Z then we interpret the string as a binary buffer.

**Parameters** **value** (*str or int or long*) – The value to assign.

**get\_value()**

Return the integer representation of the underlying vector.

**get\_value\_signed()**

Return the signed integer representation of the underlying vector.

**property is\_resolvable**

Does the value contain any X's? Inquiring minds want to know.

**property value**

Integer access to the value. **deprecated**

**property integer**

The integer representation of the underlying vector.

**property signed\_integer**

The signed integer representation of the underlying vector.

**get\_buff()**

Attribute *buff* represents the value as a binary string buffer.

```
>>> "0100000100101111".buff == "A/"
True
```

**property buff**

Access to the value as a buffer.

**get\_binstr()**

Attribute *binstr* is the binary representation stored as a string of 1 and 0.

**property binstr**

Access to the binary string.

**property n\_bits**

Access to the number of bits of the binary value.

**class cocotb.bus.Bus**(*entity*, *name*, *signals*, *optional\_signals*=[], *bus\_separator*='\_', *array\_idx*=None)

Wraps up a collection of signals.

Assumes we have a set of signals/nets named *entity*.<bus\_name><separator><signal>.

For example a bus *stream\_in* with signals *valid* and *data* is assumed to be named *dut.stream\_in\_valid* and *dut.stream\_in\_data* (with the default separator '\_').

---

**Todo:** Support for struct/record ports where signals are member names.

---

### Parameters

- **entity** (*SimHandle*) – *SimHandle* instance to the entity containing the bus.
- **name** (*str*) – Name of the bus. None for nameless bus, e.g. bus-signals in an interface or a modport (untested on struct/record, but could work here as well).
- **signals** (*list/dict*) – In the case of an obj (passed to drive/capture) that has the same attribute names as the signal names of the bus, the signals argument can be a list of those names. When obj has different attribute names, the signals arg should be a dict that maps bus attribute names to obj signal names.
- **optional\_signals** (*list/dict*, *optional*) – Signals that don't have to be present on the interface. See signals argument above for details.

- **bus\_separator** (*str*, *optional*) – Character(s) to use as separator between bus name and signal name. Defaults to ‘\_’.
- **array\_idx** (*int* or *None*, *optional*) – Optional index when signal is an array.

**drive** (*obj*, *strict=False*)

Drives values onto the bus.

**Parameters**

- **obj** – Object with attribute names that match the bus signals.
- **strict** (*bool*, *optional*) – Check that all signals are being assigned.

**Raises** **AttributeError** – If not all signals have been assigned when *strict=True*.

**capture** ()

Capture the values from the bus, returning an object representing the capture.

**Returns** A dictionary that supports access by attribute, where each attribute corresponds to each signal’s value.

**Return type** *dict*

**Raises** **RuntimeError** – If signal not present in bus, or attempt to modify a bus capture.

**sample** (*obj*, *strict=False*)

Sample the values from the bus, assigning them to *obj*.

**Parameters**

- **obj** – Object with attribute names that match the bus signals.
- **strict** (*bool*, *optional*) – Check that all signals being sampled are present in *obj*.

**Raises** **AttributeError** – If attribute is missing in *obj* when *strict=True*.

**class** cocotb.clock.Clock (*signal*, *period*, *units=None*)

Simple 50:50 duty cycle clock driver.

Instances of this class should call its *start()* method and fork the result. This will create a clocking thread that drives the signal at the desired period/frequency.

Example:

```
c = Clock(dut.clk, 10, 'ns')
cocotb.fork(c.start())
```

**Parameters**

- **signal** – The clock pin/signal to be driven.
- **period** (*int*) – The clock period. Must convert to an even number of timesteps.
- **units** (*str*, *optional*) – One of *None*, ‘fs’, ‘ps’, ‘ns’, ‘us’, ‘ms’, ‘sec’. When no *units* is given (*None*) the timestep is determined by the simulator.

## 7.3.1 Triggers

See *Simulator Triggers* for a list of sub-classes. Below are the internal classes used within cocotb.

**class** cocotb.triggers.Trigger

Base class to derive from.

**abstract prime** (*callback*)

Set a callback to be invoked when the trigger fires.

The callback will be invoked with a single argument, *self*.

Sub-classes must override this, but should end by calling the base class method.

Do not call this directly within coroutines, it is intended to be used only by the scheduler.

**unprime** ()

Remove the callback, and perform cleanup if necessary.

After being un-primed, a Trigger may be re-primed again in the future. Calling *unprime* multiple times is allowed, subsequent calls should be a no-op.

Sub-classes may override this, but should end by calling the base class method.

Do not call this directly within coroutines, it is intended to be used only by the scheduler.

**class** cocotb.triggers.GPITrigger

Base Trigger class for GPI triggers.

Consumes simulation time.

**unprime** ()

Disable a primed trigger, can be re-primed.

## 7.4 Testbench Structure

### 7.4.1 Driver

**class** cocotb.drivers.Driver

Class defining the standard interface for a driver within a testbench.

The driver is responsible for serializing transactions onto the physical pins of the interface. This may consume simulation time.

Constructor for a driver instance.

**kill** ()

Kill the coroutine sending stuff.

**append** (*transaction*, *callback=None*, *event=None*, *\*\*kwargs*)

Queue up a transaction to be sent over the bus.

Mechanisms are provided to permit the caller to know when the transaction is processed.

#### Parameters

- **transaction** (*any*) – The transaction to be sent.
- **callback** (*callable*, *optional*) – Optional function to be called when the transaction has been sent.
- **event** (*optional*) – *Event* to be set when the transaction has been sent.
- **\*\*kwargs** – Any additional arguments used in child class' *\_driver\_send* method.

**clear** ()

Clear any queued transactions without sending them onto the bus.

**send**

Blocking send call (hence must be “yielded” rather than called).

Sends the transaction over the bus.

**Parameters**

- **transaction** (*any*) – The transaction to be sent.
- **sync** (*bool*, *optional*) – Synchronize the transfer by waiting for a rising edge.
- **\*\*kwargs** (*dict*) – Additional arguments used in child class’ `_driver_send` method.

**\_driver\_send** (*transaction*, *sync=True*, *\*\*kwargs*)

Actual implementation of the send.

Sub-classes should override this method to implement the actual `send()` routine.

**Parameters**

- **transaction** (*any*) – The transaction to be sent.
- **sync** (*boolean*, *optional*) – Synchronize the transfer by waiting for a rising edge.
- **\*\*kwargs** – Additional arguments if required for protocol implemented in a sub-class.

**\_send**

Send coroutine.

**Parameters**

- **transaction** (*any*) – The transaction to be sent.
- **callback** (*callable*, *optional*) – Optional function to be called when the transaction has been sent.
- **event** (*optional*) – event to be set when the transaction has been sent.
- **sync** (*boolean*, *optional*) – Synchronize the transfer by waiting for a rising edge.
- **\*\*kwargs** – Any additional arguments used in child class’ `_driver_send` method.

**class** cocotb.drivers.BitDriver (*signal*, *clk*, *generator=None*)

Bases: `object`

Drives a signal onto a single bit.

Useful for exercising ready/valid flags.

**start** (*generator=None*)

Start generating data.

**Parameters** **generator** (*generator*, *optional*) – Generator yielding data. The generator should yield tuples (`on`, `off`) with the number of cycles to be on, followed by the number of cycles to be off. Typically the generator should go on forever.

Example:

```
bit_driver.start((1, i % 5) for i in itertools.count())
```

**stop** ()

Stop generating data.

**class** cocotb.drivers.BusDriver (*entity*, *name*, *clock*, *\*\*kwargs*)

Bases: `cocotb.drivers.Driver`

Wrapper around common functionality for buses which have:

- a list of `_signals` (class attribute)
- a list of `_optional_signals` (class attribute)
- a clock
- a name
- an entity

#### Parameters

- **entity** (*SimHandle*) – A handle to the simulator entity.
- **name** (*str* or *None*) – Name of this bus. *None* for a nameless bus, e.g. bus-signals in an interface or a modport. (untested on *struct/record*, but could work here as well).
- **clock** (*SimHandle*) – A handle to the clock associated with this bus.
- **array\_idx** (*int* or *None*, *optional*) – Optional index when signal is an array.

Constructor for a driver instance.

#### `_driver_send`

Implementation for *BusDriver*.

#### Parameters

- **transaction** – The transaction to send.
- **sync** (*bool*, *optional*) – Synchronize the transfer by waiting for a rising edge.

#### `_wait_for_signal`

This method will return when the specified signal has hit logic 1. The state will be in the *ReadOnly* phase so sim will need to move to *NextTimeStep* before registering more callbacks can occur.

#### `_wait_for_nsignal`

This method will return when the specified signal has hit logic 0. The state will be in the *ReadOnly* phase so sim will need to move to *NextTimeStep* before registering more callbacks can occur.

**class** `cocotb.drivers.ValidatedBusDriver` (*entity*, *name*, *clock*, *\*\*kwargs*)

Bases: `cocotb.drivers.BusDriver`

Same as a *BusDriver* except we support an optional generator to control which cycles are valid.

#### Parameters

- **entity** (*SimHandle*) – A handle to the simulator entity.
- **name** (*str*) – Name of this bus.
- **clock** (*SimHandle*) – A handle to the clock associated with this bus.
- **valid\_generator** (*generator*, *optional*) – a generator that yields tuples of (valid, invalid) cycles to insert.

Constructor for a driver instance.

#### `_next_valids` ()

Optionally insert invalid cycles every N cycles.

The generator should yield tuples with the number of cycles to be on followed by the number of cycles to be off. The on cycles should be non-zero, we skip invalid generator entries.

**set\_valid\_generator** (*valid\_generator=None*)  
Set a new valid generator for this bus.

## 7.4.2 Monitor

**class** cocotb.monitors.**Monitor** (*callback=None, event=None*)  
Base class for Monitor objects.

Monitors are passive ‘listening’ objects that monitor pins going in or out of a DUT. This class should not be used directly, but should be sub-classed and the internal `_monitor_recv` method should be overridden and decorated as a *coroutine*. This `_monitor_recv` method should capture some behavior of the pins, form a transaction, and pass this transaction to the internal `_recv` method. The `_monitor_recv` method is added to the cocotb scheduler during the `__init__` phase, so it should not be yielded anywhere.

The primary use of a Monitor is as an interface for a *Scoreboard*.

### Parameters

- **callback** (*callable*) – Callback to be called with each recovered transaction as the argument. If the callback isn’t used, received transactions will be placed on a queue and the event used to notify any consumers.
- **event** (*cocotb.triggers.Event*) – Event that will be called when a transaction is received through the internal `_recv` method. *Event.data* is set to the received transaction.

**wait\_for\_recv** (*timeout=None*)

With *timeout*, *wait()* for transaction to arrive on monitor and return its data.

**Parameters** *timeout* – The timeout value for *Timer*. Defaults to None.

**Returns** Data of received transaction.

**\_monitor\_recv**

Actual implementation of the receiver.

Sub-classes should override this method to implement the actual receive routine and call `_recv` with the recovered transaction.

**\_recv** (*transaction*)

Common handling of a received transaction.

**class** cocotb.monitors.**BusMonitor** (*entity, name, clock, reset=None, reset\_n=None, callback=None, event=None, bus\_separator='\_', array\_idx=None*)

Bases: *cocotb.monitors.Monitor*

Wrapper providing common functionality for monitoring buses.

**property in\_reset**

Boolean flag showing whether the bus is in reset state or not.

## 7.4.3 Scoreboard

Common scoreboarding capability.

**class** cocotb.scoreboard.**Scoreboard** (*dut, reorder\_depth=0, fail\_immediately=True*)  
Bases: *object*

Generic scoreboarding class.

We can add interfaces by providing a monitor and an expected output queue.



The expected output can either be a function which provides a transaction or a simple list containing the expected output.

---

**Todo:** Statistics for end-of-test summary etc.

---

### Parameters

- **dut** (*SimHandle*) – Handle to the DUT.
- **reorder\_depth** (*int*, *optional*) – Consider up to *reorder\_depth* elements of the expected result list as passing matches. Default is 0, meaning only the first element in the expected result list is considered for a passing match.
- **fail\_immediately** (*bool*, *optional*) – Raise *TestFailure* immediately when something is wrong instead of just recording an error. Default is *True*.

### property result

Determine the test result, do we have any pending data remaining?

**Returns** If not all expected output was received or error were recorded during the test.

**Return type** *TestFailure*

**compare** (*got*, *exp*, *log*, *strict\_type=True*)

Common function for comparing two transactions.

Can be re-implemented by a sub-class.

### Parameters

- **got** – The received transaction.
- **exp** – The expected transaction.
- **log** – The logger for reporting messages.
- **strict\_type** (*bool*, *optional*) – Require transaction type to match exactly if *True*, otherwise compare its string representation.

**Raises** *TestFailure* – If received transaction differed from expected transaction when *fail\_immediately* is *True*. If *strict\_type* is *True*, also the transaction type must match.

**add\_interface** (*monitor*, *expected\_output*, *compare\_fn=None*, *reorder\_depth=0*, *strict\_type=True*)

Add an interface to be scoreboarded.

Provides a function which the monitor will callback with received transactions.

Simply check against the expected output.

### Parameters

- **monitor** – The monitor object.
- **expected\_output** – Queue of expected outputs.
- **compare\_fn** (*callable*, *optional*) – Function doing the actual comparison.
- **reorder\_depth** (*int*, *optional*) – Consider up to *reorder\_depth* elements of the expected result list as passing matches. Default is 0, meaning only the first element in the expected result list is considered for a passing match.

- **strict\_type** (*bool*, *optional*) – Require transaction type to match exactly if True, otherwise compare its string representation.

Raises **TypeError** – If no monitor is on the interface or *compare\_fn* is not a callable function.

## 7.4.4 Clock

**class** cocotb.clock.Clock (*signal*, *period*, *units=None*)

Simple 50:50 duty cycle clock driver.

Instances of this class should call its *start()* method and fork the result. This will create a clocking thread that drives the signal at the desired period/frequency.

Example:

```
c = Clock(dut.clk, 10, 'ns')
cocotb.fork(c.start())
```

### Parameters

- **signal** – The clock pin/signal to be driven.
- **period** (*int*) – The clock period. Must convert to an even number of timesteps.
- **units** (*str*, *optional*) – One of None, 'fs', 'ps', 'ns', 'us', 'ms', 'sec'. When no *units* is given (None) the timestep is determined by the simulator.

### start

Clocking coroutine. Start driving your clock by forking a call to this.

### Parameters

- **cycles** (*int*, *optional*) – Cycle the clock *cycles* number of times, or if None then cycle the clock forever. Note: 0 is not the same as None, as 0 will cycle no times.
- **start\_high** (*bool*, *optional*) – Whether to start the clock with a 1 for the first half of the period. Default is True.

## 7.5 Utilities

cocotb.utils.get\_sim\_time (*units=None*)

Retrieves the simulation time from the simulator.

**Parameters** **units** (*str* or *None*, *optional*) – String specifying the units of the result (one of None, 'fs', 'ps', 'ns', 'us', 'ms', 'sec'). None will return the raw simulation time.

**Returns** The simulation time in the specified units.

cocotb.utils.get\_time\_from\_sim\_steps (*steps*, *units*)

Calculates simulation time in the specified *units* from the *steps* based on the simulator precision.

### Parameters

- **steps** (*int*) – Number of simulation steps.
- **units** (*str*) – String specifying the units of the result (one of 'fs', 'ps', 'ns', 'us', 'ms', 'sec').

**Returns** The simulation time in the specified units.

`cocotb.utils.get_sim_steps(time, units=None)`

Calculates the number of simulation time steps for a given amount of *time*.

**Parameters**

- **time** (*numbers.Number*) – The value to convert to simulation time steps.
- **units** (*str or None, optional*) – String specifying the units of the result (one of `None`, `'fs'`, `'ps'`, `'ns'`, `'us'`, `'ms'`, `'sec'`). `None` means time is already in simulation time steps.

**Returns** The number of simulation time steps.

**Return type** `int`

**Raises** **ValueError** – If given *time* cannot be represented by simulator precision.

`cocotb.utils.pack(ctypes_obj)`

Convert a `ctypes` structure into a Python string.

**Parameters** **ctypes\_obj** (*ctypes.Structure*) – The `ctypes` structure to convert to a string.

**Returns** New Python string containing the bytes from memory holding *ctypes\_obj*.

`cocotb.utils.unpack(ctypes_obj, string, bytes=None)`

Unpack a Python string into a `ctypes` structure.

If the length of *string* is not the correct size for the memory footprint of the `ctypes` structure then the *bytes* keyword argument must be used.

**Parameters**

- **ctypes\_obj** (*ctypes.Structure*) – The `ctypes` structure to pack into.
- **string** (*str*) – String to copy over the *ctypes\_obj* memory space.
- **bytes** (*int, optional*) – Number of bytes to copy. Defaults to `None`, meaning the length of *string* is used.

**Raises**

- **ValueError** – If length of *string* and size of *ctypes\_obj* are not equal.
- **MemoryError** – If *bytes* is longer than size of *ctypes\_obj*.

`cocotb.utils.hexdump(x)`

Hexdump a buffer.

**Parameters** **x** – Object that supports conversion via the `str` built-in.

**Returns** A string containing the hexdump.

## Example

```
>>> print(hexdump('this somewhat long string'))
0000  74 68 69 73 20 73 6F 6D 65 77 68 61 74 20 6C 6F  this somewhat lo
0010  6E 67 20 73 74 72 69 6E 67                          ng string
```

`cocotb.utils.hexdiffs(x, y)`

Return a diff string showing differences between two binary strings.

**Parameters**

- **x** – Object that supports conversion via the `str` built-in.
- **y** – Object that supports conversion via the `str` built-in.

### Example

```
>>> print(hexdiffs(0, 1))
0000      30                                0
      0000 31                                1

>>> print(hexdiffs('a', 'b'))
0000      61                                a
      0000 62                                b

>>> print(hexdiffs('this short thing', 'this also short'))
0000      746869732073686F 7274207468696E67 this short thing
      0000 7468697320616C73 6F 2073686F7274 this also short
```

**class** `cocotb.utils.ParametrizedSingleton(*args, **kwargs)`

A metaclass that allows class construction to reuse an existing instance.

We use this so that `RisingEdge(sig)` and `Join(coroutine)` always return the same instance, rather than creating new copies.

`cocotb.utils.reject_remaining_kwargs(name, kwargs)`

Helper function to emulate Python 3 keyword-only arguments.

Use as:

```
def func(x1, **kwargs):
    a = kwargs.pop('a', 1)
    b = kwargs.pop('b', 2)
    reject_remaining_kwargs('func', kwargs)
    ...
```

To emulate the Python 3 syntax:

```
def func(x1, *, a=1, b=2):
    ...
```

**class** `cocotb.utils.lazy_property(fget)`

A property that is executed the first time, then cached forever.

It does this by replacing itself on the instance, which works because unlike `@property` it does not define `__set__`.

This should be used for expensive members of objects that are not always used.

`cocotb.utils.want_color_output()`

Return True if colored output is possible/requested and not running in GUI.

`cocotb.utils.remove_traceback_frames(tb_or_exc, frame_names)`

Strip leading frames from a traceback

#### Parameters

- **tb\_or\_exc** (`Union[traceback, BaseException, exc_info]`) – Object to strip frames from. If an exception is passed, creates a copy of the exception with a new shorter traceback. If a tuple from `sys.exc_info` is passed, returns the same tuple with the traceback shortened

- **frame\_names** (*List[str]*) – Names of the frames to strip, which must be present.

## 7.6 Simulation Object Handles

**class** cocotb.handle.SimHandleBase (*handle, path*)

Bases: *object*

Base class for all simulation objects.

We maintain a handle which we can use for GPI calls.

### Parameters

- **handle** (*integer*) – the GPI handle to the simulator object
- **path** (*string*) – path to this handle, None if root

**class** cocotb.handle.RegionObject (*handle, path*)

Bases: *cocotb.handle.SimHandleBase*

Region objects don't have values, they are effectively scopes or namespaces.

Args: handle (*integer*) : the GPI handle to the simulator object path (*string*) : path to this handle, None if root

**class** cocotb.handle.HierarchyObject (*handle, path*)

Bases: *cocotb.handle.RegionObject*

Hierarchy objects are namespace/scope objects.

Args: handle (*integer*) : the GPI handle to the simulator object path (*string*) : path to this handle, None if root

**class** cocotb.handle.HierarchyArrayObject (*handle, path*)

Bases: *cocotb.handle.RegionObject*

Hierarchy Arrays are containers of Hierarchy Objects.

Args: handle (*integer*) : the GPI handle to the simulator object path (*string*) : path to this handle, None if root

**class** cocotb.handle.AssignmentResult (*signal, value*)

Bases: *object*

Object that exists solely to provide an error message if the caller is not aware of cocotb's meaning of <=.

**class** cocotb.handle.NonHierarchyObject (*handle, path*)

Bases: *cocotb.handle.SimHandleBase*

Common base class for all non-hierarchy objects.

Args: handle (*integer*) : the GPI handle to the simulator object path (*string*) : path to this handle, None if root

### property value

A reference to the value

**class** cocotb.handle.ConstantObject (*handle, path, handle\_type*)

Bases: *cocotb.handle.NonHierarchyObject*

Constant objects have a value that can be read, but not set.

We can also cache the value since it is elaboration time fixed and won't change within a simulation.

Args: handle (*integer*) : the GPI handle to the simulator object path (*string*) : path to this handle, None if root

**class** cocotb.handle.**NonHierarchyIndexableObject** (*handle, path*)

Bases: *cocotb.handle.NonHierarchyObject*

Args: handle (int): FLI/VPI/VHPI handle to the simulator object.

**class** cocotb.handle.**NonConstantObject** (*handle, path*)

Bases: *cocotb.handle.NonHierarchyIndexableObject*

Args: handle (int): FLI/VPI/VHPI handle to the simulator object.

**drivers** ()

An iterator for gathering all drivers for a signal.

**loads** ()

An iterator for gathering all loads on a signal.

**class** cocotb.handle.**ModifiableObject** (*handle, path*)

Bases: *cocotb.handle.NonConstantObject*

Base class for simulator objects whose values can be modified.

Args: handle (int): FLI/VPI/VHPI handle to the simulator object.

**setimmediatevalue** (*value*)

Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

We determine the library call to make based on the type of the value because assigning integers less than 32 bits is faster.

**Parameters value** (*ctypes.Structure, cocotb.binary.BinaryValue, int, double*) – The value to drive onto the simulator object.

**Raises *TypeError*** – If target is not wide enough or has an unsupported type for value assignment.

**class** cocotb.handle.**RealObject** (*handle, path*)

Bases: *cocotb.handle.ModifiableObject*

Specific object handle for Real signals and variables.

Args: handle (int): FLI/VPI/VHPI handle to the simulator object.

**setimmediatevalue** (*value*)

Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

**Parameters value** (*float*) – The value to drive onto the simulator object.

**Raises *TypeError*** – If target has an unsupported type for real value assignment.

**class** cocotb.handle.**EnumObject** (*handle, path*)

Bases: *cocotb.handle.ModifiableObject*

Specific object handle for enumeration signals and variables.

Args: handle (int): FLI/VPI/VHPI handle to the simulator object.

**setimmediatevalue** (*value*)

Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

**Parameters value** (*int*) – The value to drive onto the simulator object.

**Raises** `TypeError` – If target has an unsupported type for integer value assignment.

**class** `cocotb.handle.IntegerObject` (*handle*, *path*)

Bases: `cocotb.handle.ModifiableObject`

Specific object handle for Integer and Enum signals and variables.

Args: *handle* (*int*): FLI/VPI/VHPI handle to the simulator object.

**setimmediatevalue** (*value*)

Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

**Parameters** *value* (*int*) – The value to drive onto the simulator object.

**Raises** `TypeError` – If target has an unsupported type for integer value assignment.

**class** `cocotb.handle.StringObject` (*handle*, *path*)

Bases: `cocotb.handle.ModifiableObject`

Specific object handle for String variables.

Args: *handle* (*int*): FLI/VPI/VHPI handle to the simulator object.

**setimmediatevalue** (*value*)

Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

**Parameters** *value* (*str*) – The value to drive onto the simulator object.

**Raises** `TypeError` – If target has an unsupported type for string value assignment.

`cocotb.handle.SimHandle` (*handle*, *path=None*)

Factory function to create the correct type of `SimHandle` object.

## 7.7 Implemented Testbench Structures

### 7.7.1 Drivers

#### AD9361

Analog Devices AD9361 RF Transceiver.

```
class cocotb.drivers.ad9361.AD9361 (dut, rx_channels=1, tx_channels=1,
                                     tx_clock_half_period=16276,
                                     rx_clock_half_period=16276, loop-
                                     back_queue_maxlen=16)
```

Driver for the AD9361 RF Transceiver.

Constructor for a driver instance.

```
send_data (i_data, q_data, i_data2=None, q_data2=None, binaryRepresenta-
            tion=BinaryRepresentation.TWOS_COMPLEMENT)
```

Forks the `rx_data_to_ad9361` coroutine to send data.

**Parameters**

- **i\_data** (*int*) – Data of the I0 channel.
- **q\_data** (*int*) – Data of the Q0 channel.

- **i\_data2** (*int*, *optional*) – Data of the I1 channel.
- **q\_data2** (*int*, *optional*) – Data of the Q1 channel.
- **binaryRepresentation** ([BinaryRepresentation](#)) – The representation of the binary value. Default is `TWOS_COMPLEMENT`.

**rx\_data\_to\_ad9361** (*i\_data*, *q\_data*, *i\_data2=None*, *q\_data2=None*, *binaryRepresentation=BinaryRepresentation.TWOS\_COMPLEMENT*)  
Receive data to AD9361.

This is a coroutine.

#### Parameters

- **i\_data** (*int*) – Data of the I0 channel.
- **q\_data** (*int*) – Data of the Q0 channel.
- **i\_data2** (*int*, *optional*) – Data of the I1 channel.
- **q\_data2** (*int*, *optional*) – Data of the Q1 channel.
- **binaryRepresentation** ([BinaryRepresentation](#)) – The representation of the binary value. Default is `TWOS_COMPLEMENT`.

**ad9361\_tx\_to\_rx\_loopback** ()

Create loopback from tx to rx.

Forks a coroutine doing the actual task.

**tx\_data\_from\_ad9361** ()

Transmit data from AD9361.

Forks a coroutine doing the actual task.

## AMBA

Advanced Microcontroller Bus Architecture.

**class** `cocotb.drivers.amba.AXI4LiteMaster` (*entity*, *name*, *clock*)  
AXI4-Lite Master.

TODO: Kill all pending transactions if reset is asserted.

Constructor for a driver instance.

**write** (*address*, *value*, *byte\_enable=0xf*, *address\_latency=0*, *data\_latency=0*)  
Write a value to an address.

#### Parameters

- **address** (*int*) – The address to write to.
- **value** (*int*) – The data value to write.
- **byte\_enable** (*int*, *optional*) – Which bytes in value to actually write. Default is to write all bytes.
- **address\_latency** (*int*, *optional*) – Delay before setting the address (in clock cycles). Default is no delay.
- **data\_latency** (*int*, *optional*) – Delay before setting the data value (in clock cycles). Default is no delay.
- **sync** (*bool*, *optional*) – Wait for rising edge on clock initially. Defaults to True.



**Returns** The write response value.

**Return type** *BinaryValue*

**Raises** **AXIProtocolError** – If write response from AXI is not OKAY.

**read** (*address*, *sync=True*)

Read from an address.

**Parameters**

- **address** (*int*) – The address to read from.
- **sync** (*bool*, *optional*) – Wait for rising edge on clock initially. Defaults to True.

**Returns** The read data value.

**Return type** *BinaryValue*

**Raises** **AXIProtocolError** – If read response from AXI is not OKAY.

```
class cocotb.drivers.amba.AXI4Slave (entity, name, clock, memory, callback=None,  
                                     event=None, big_endian=False)
```

AXI4 Slave

Monitors an internal memory and handles read and write requests.

Constructor for a driver instance.

## Avalon

```
class cocotb.drivers.avalon.AvalonMM (entity, name, clock, **kwargs)
```

Bases: *cocotb.drivers.BusDriver*

Avalon Memory Mapped Interface (Avalon-MM) Driver.

Currently we only support the mode required to communicate with SF `avalon_mapper` which is a limited subset of all the signals.

Blocking operation is all that is supported at the moment, and for the near future as well. Posted responses from a slave are not supported.

Constructor for a driver instance.

```
class cocotb.drivers.avalon.AvalonMaster (entity, name, clock, **kwargs)
```

Avalon Memory Mapped Interface (Avalon-MM) Master.

Constructor for a driver instance.

**write** (*address*, *value*)

Issue a write to the given address with the specified value.

**Parameters**

- **address** (*int*) – The address to write to.
- **value** (*int*) – The data value to write.

**Raises** **TestError** – If master is read-only.

**read** (*address*, *sync=True*)

Issue a request to the bus and block until this comes back.

Simulation time still progresses but syntactically it blocks.

**Parameters**

- **address** (*int*) – The address to read from.
- **sync** (*bool*, *optional*) – Wait for rising edge on clock initially. Defaults to True.

**Returns** The read data value.

**Return type** *BinaryValue*

**Raises** *TestError* – If master is write-only.

```
class cocotb.drivers.avalon.AvalonMemory(entity, name, clock, readlatency_min=1,
                                         readlatency_max=1, memory=None,
                                         avl_properties={})
```

Bases: *cocotb.drivers.BusDriver*

Emulate a memory, with back-door access.

Constructor for a driver instance.

```
class cocotb.drivers.avalon.AvalonST(*args, **kwargs)
```

Bases: *cocotb.drivers.ValidatedBusDriver*

Avalon Streaming Interface (Avalon-ST) Driver

Constructor for a driver instance.

```
class cocotb.drivers.avalon.AvalonSTPkts(*args, **kwargs)
```

Bases: *cocotb.drivers.ValidatedBusDriver*

Avalon Streaming Interface (Avalon-ST) Driver, packetized.

Constructor for a driver instance.

## OPB

```
class cocotb.drivers.opb.OPBMaster(entity, name, clock)
```

On-chip peripheral bus master.

Constructor for a driver instance.

```
write(address, value, sync=True)
```

Issue a write to the given address with the specified value.

### Parameters

- **address** (*int*) – The address to read from.
- **value** (*int*) – The data value to write.
- **sync** (*bool*, *optional*) – Wait for rising edge on clock initially. Defaults to True.

**Raises** *OPBException* – If write took longer than 16 cycles.

```
read(address, sync=True)
```

Issue a request to the bus and block until this comes back.

Simulation time still progresses but syntactically it blocks.

### Parameters

- **address** (*int*) – The address to read from.
- **sync** (*bool*, *optional*) – Wait for rising edge on clock initially. Defaults to True.

**Returns** The read data value.

**Return type** *BinaryValue*

**Raises** `OPBException` – If read took longer than 16 cycles.

## XGMII

**class** `cocotb.drivers.xgmii.XGMII` (*signal, clock, interleaved=True*)

Bases: `cocotb.drivers.Driver`

XGMII (10 Gigabit Media Independent Interface) driver.

### Parameters

- **signal** (*SimHandle*) – The XGMII data bus.
- **clock** (*SimHandle*) – The associated clock (assumed to be driven by another coroutine).
- **interleaved** (*bool, optional*) – Whether control bits are interleaved with the data bytes or not.

**If interleaved the bus is** `byte0, byte0_control, byte1, byte1_control, ...`

**Otherwise expect** `byte0, byte1, ..., byte0_control, byte1_control, ...`

**static** `layer1` (*packet*)

Take an Ethernet packet (as a string) and format as a layer 1 packet.

Pad to 64 bytes, prepend preamble and append 4-byte CRC on the end.

**Parameters** `packet` (*str*) – The Ethernet packet to format.

**Returns** The formatted layer 1 packet.

**Return type** `str`

`idle()`

Helper function to set bus to IDLE state.

`terminate` (*index*)

Helper function to terminate from a provided lane index.

**Parameters** `index` (*int*) – The index to terminate.

## 7.7.2 Monitors

### Avalon

**class** `cocotb.monitors.avalon.AvalonST` (*\*args, \*\*kwargs*)

Bases: `cocotb.monitors.BusMonitor`

Avalon-ST bus.

Non-packetized so each valid word is a separate transaction.

**class** `cocotb.monitors.avalon.AvalonSTPkts` (*\*args, \*\*kwargs*)

Bases: `cocotb.monitors.BusMonitor`

Packetized Avalon-ST bus.

## XGMII

**class** cocotb.monitors.xgmii.**XGMII**(*signal*, *clock*, *interleaved=True*, *callback=None*,  
*event=None*)

Bases: *cocotb.monitors.Monitor*

XGMII (10 Gigabit Media Independent Interface) Monitor.

Assumes a single vector, either 4 or 8 bytes plus control bit for each byte.

If interleaved is `True` then the control bits are adjacent to the bytes.

### Parameters

- **signal** (*SimHandle*) – The XGMII data bus.
- **clock** (*SimHandle*) – The associated clock (assumed to be driven by another coroutine).
- **interleaved** (*bool*, *optional*) – Whether control bits are interleaved with the data bytes or not.

If interleaved the bus is `byte0, byte0_control, byte1, byte1_control, ...`

Otherwise expect `byte0, byte1, ..., byte0_control, byte1_control, ...`

## 7.8 Miscellaneous

### 7.8.1 Signal Tracer for WaveDrom

**class** cocotb.wavedrom.**Wavedrom**(*obj*)  
Base class for a WaveDrom compatible tracer.

**sample**()  
Record a sample of the signal value at this point in time.

**clear**()  
Delete all sampled data.

**get**(*add\_clock=True*)  
Return the samples as a list suitable for use with WaveDrom.

**class** cocotb.wavedrom.**trace**(\*args, \*\*kwargs)  
Context manager to enable tracing of signals.

Arguments are an arbitrary number of signals or buses to trace. We also require a clock to sample on, passed in as a keyword argument.

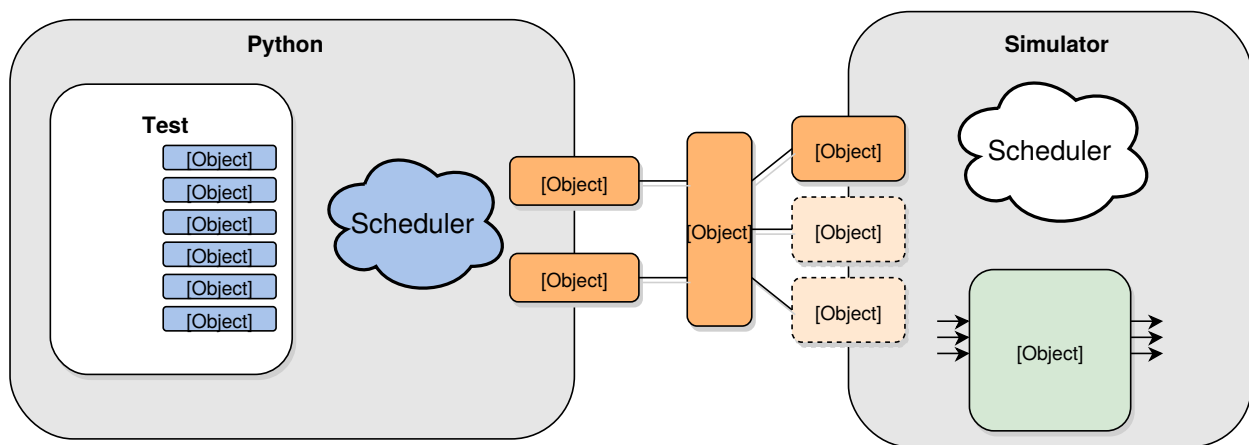
Usage:

```
with trace(sig1, sig2, a_bus, clk=clk) as waves:
    # Stuff happens, we trace it

    # Dump to JSON format compatible with WaveDrom
    j = waves.dumpj()
```

## C CODE LIBRARY REFERENCE

Cocotb contains a library called GPI (in directory `cocotb/share/lib/gpi/`) written in C++ that is an abstraction layer for the VPI, VHPI, and FLI simulator interfaces.



The interaction between Python and GPI is via a Python extension module called `simulator` (in directory `cocotb/share/lib/simulator/`) which provides routines for traversing the hierarchy, getting/setting an object's value, registering callbacks etc.

## 8.1 API Documentation

### 8.1.1 Class list

#### Class `FliEnumObjHdl`

```
class FliEnumObjHdl : public FliValueObjHdl
```

#### Class `FliImpl`

```
class FliImpl : public GpiImplInterface
```

#### Native Check Create

Determine whether a simulation object is native to FLI and create a handle if it is

### Get current simulation time

Get current simulation time

NB units depend on the simulation configuration

### Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

### Class FliIntObjHdl

```
class FliIntObjHdl : public FliValueObjHdl
```

### Class FliIterator

```
class FliIterator : public GpiIterator
```

### Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

### Class FliLogicObjHdl

```
class FliLogicObjHdl : public FliValueObjHdl
```

### Class FliNextPhaseCbHdl

```
class FliNextPhaseCbHdl : public FliSimPhaseCbHdl
```

### Class FliObj

```
class FliObj
```

Subclassed by *FliObjHdl*, *FliSignalObjHdl*

### Class FliObjHdl

```
class FliObjHdl : public GpiObjHdl, public FliObj
```

### Class `FliProcessCbHdl`

**class** `FliProcessCbHdl` : **public virtual** *GpiCbHdl*

Subclassed by *FliShutdownCbHdl*, *FliSignalCbHdl*, *FliSimPhaseCbHdl*, *FliStartupCbHdl*, *FliTimedCbHdl*

#### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

### Class `FliReadOnlyCbHdl`

**class** `FliReadOnlyCbHdl` : **public** *FliSimPhaseCbHdl*

### Class `FliReadWriteCbHdl`

**class** `FliReadWriteCbHdl` : **public** *FliSimPhaseCbHdl*

### Class `FliRealObjHdl`

**class** `FliRealObjHdl` : **public** *FliValueObjHdl*

### Class `FliShutdownCbHdl`

**class** `FliShutdownCbHdl` : **public** *FliProcessCbHdl*

#### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

### Class `FliSignalCbHdl`

**class** `FliSignalCbHdl` : **public** *FliProcessCbHdl*, **public** *GpiValueCbHdl*

#### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

### Class `FliSignalObjHdl`

```
class FliSignalObjHdl : public GpiSignalObjHdl, public FliObj  
    Subclassed by FliValueObjHdl
```

### Class `FliSimPhaseCbHdl`

```
class FliSimPhaseCbHdl : public FliProcessCbHdl  
    Subclassed by FliNextPhaseCbHdl, FliReadOnlyCbHdl, FliReadWriteCbHdl
```

#### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized. . .

### Class `FliStartupCbHdl`

```
class FliStartupCbHdl : public FliProcessCbHdl
```

#### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized. . .

### Class `FliStringObjHdl`

```
class FliStringObjHdl : public FliValueObjHdl
```

### Class `FliTimedCbHdl`

```
class FliTimedCbHdl : public FliProcessCbHdl
```

#### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized. . .

### Class `FliTimerCache`

```
class FliTimerCache
```



## Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

## Class FliValueObjHdl

**class FliValueObjHdl** : public *FliSignalObjHdl*

Subclassed by *FliEnumObjHdl*, *FliIntObjHdl*, *FliLogicObjHdl*, *FliRealObjHdl*, *FliStringObjHdl*

## Class GpiCbHdl

**class GpiCbHdl** : public *GpiHdl*

Subclassed by *FliProcessCbHdl*, *GpiValueCbHdl*, *VhpiCbHdl*, *VpiCbHdl*

## Class GpiClockHdl

**class GpiClockHdl**

## Class GpiHdl

**class GpiHdl**

Subclassed by *GpiCbHdl*, *GpiIterator*, *GpiObjHdl*

## Class GpiImplInterface

**class GpiImplInterface**

Subclassed by *FliImpl*, *VhpiImpl*, *VpiImpl*

## Class GpiIterator

**class GpiIterator** : public *GpiHdl*

Subclassed by *FliIterator*, *VhpiIterator*, *VpiIterator*, *VpiSingleIterator*

## Class GpiIteratorMapping

template<class **Ti**, class **Tm**>

**class GpiIteratorMapping**

## Class GpiObjHdl

**class GpiObjHdl** : public *GpiHdl*

Subclassed by *FliObjHdl*, *GpiSignalObjHdl*, *VhpiArrayObjHdl*, *VhpiObjHdl*, *VpiArrayObjHdl*, *VpiObjHdl*

**Class GpiSignalObjHdl**

```
class GpiSignalObjHdl : public GpiObjHdl
```

Subclassed by *FliSignalObjHdl*, *VhpiSignalObjHdl*, *VpiSignalObjHdl*

**Class GpiValueCbHdl**

```
class GpiValueCbHdl : public virtual GpiCbHdl
```

Subclassed by *FliSignalCbHdl*, *VhpiValueCbHdl*, *VpiValueCbHdl*

**Class VhpiArrayObjHdl**

```
class VhpiArrayObjHdl : public GpiObjHdl
```

**Class VhpiCbHdl**

```
class VhpiCbHdl : public virtual GpiCbHdl
```

Subclassed by *VhpiNextPhaseCbHdl*, *VhpiReadOnlyCbHdl*, *VhpiReadwriteCbHdl*, *VhpiShutdownCbHdl*, *VhpiStartupCbHdl*, *VhpiTimedCbHdl*, *VhpiValueCbHdl*

**Class VhpiImpl**

```
class VhpiImpl : public GpiImplInterface
```

**Class VhpiIterator**

```
class VhpiIterator : public GpiIterator
```

**Class VhpiLogicSignalObjHdl**

```
class VhpiLogicSignalObjHdl : public VhpiSignalObjHdl
```

**Class VhpiNextPhaseCbHdl**

```
class VhpiNextPhaseCbHdl : public VhpiCbHdl
```

**Class VhpiObjHdl**

```
class VhpiObjHdl : public GpiObjHdl
```

**Class VhpiReadOnlyCbHdl**

```
class VhpiReadOnlyCbHdl : public VhpiCbHdl
```

**Class VhpiReadwriteCbHdl**

```
class VhpiReadwriteCbHdl : public VhpiCbHdl
```

**Class VhpiShutdownCbHdl**

```
class VhpiShutdownCbHdl : public VhpiCbHdl
```

**Class VhpiSignalObjHdl**

```
class VhpiSignalObjHdl : public GpiSignalObjHdl
    Subclassed by VhpiLogicSignalObjHdl
```

**Class VhpiStartupCbHdl**

```
class VhpiStartupCbHdl : public VhpiCbHdl
```

**Class VhpiTimedCbHdl**

```
class VhpiTimedCbHdl : public VhpiCbHdl
```

**Class VhpiValueCbHdl**

```
class VhpiValueCbHdl : public VhpiCbHdl, public GpiValueCbHdl
```

**Class VpiArrayObjHdl**

```
class VpiArrayObjHdl : public GpiObjHdl
```

**Class VpiCbHdl**

```
class VpiCbHdl : public virtual GpiCbHdl
    Subclassed by VpiNextPhaseCbHdl, VpiReadOnlyCbHdl, VpiReadwriteCbHdl, VpiShutdownCbHdl, VpiStar-
    tupCbHdl, VpiTimedCbHdl, VpiValueCbHdl
```

**Class VpiImpl**

```
class VpiImpl : public GpiImplInterface
```

**Class Vpiterator**

```
class VpiIterator : public GpiIterator
```

**Class VpiNextPhaseCbHdl**

```
class VpiNextPhaseCbHdl : public VpiCbHdl
```

**Class VpiObjHdl**

```
class VpiObjHdl : public GpiObjHdl
```

### Class VpiReadOnlyCbHdl

```
class VpiReadOnlyCbHdl : public VpiCbHdl
```

### Class VpiReadwriteCbHdl

```
class VpiReadwriteCbHdl : public VpiCbHdl
```

### Class VpiShutdownCbHdl

```
class VpiShutdownCbHdl : public VpiCbHdl
```

### Class VpiSignalObjHdl

```
class VpiSignalObjHdl : public GpiSignalObjHdl
```

### Class VpiSingleIterator

```
class VpiSingleIterator : public GpiIterator
```

### Class VpiStartupCbHdl

```
class VpiStartupCbHdl : public VpiCbHdl
```

### Class VpiTimedCbHdl

```
class VpiTimedCbHdl : public VpiCbHdl
```

### Class VpiValueCbHdl

```
class VpiValueCbHdl : public VpiCbHdl, public GpiValueCbHdl
```

### Class cocotb\_entrypoint

```
class cocotb_entrypoint
```

### Class cocotb\_entrypoint::cocotb\_arch

```
class cocotb_arch
```

## 8.1.2 File list

### File FliCbHdl.cpp

### File FliImpl.cpp

### Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

```
void fli_mappings (GpiteratorMapping<int, Fliiterator::OneToMany> &map)
```

```
void handle_fli_callback (void *data)
```

```
static void register_initial_callback ()
```

```
static void register_final_callback ()
```

```
static void register_embed ()
```

```
void cocotb_init ()
```

```
GPI_ENTRY_POINT (fli, register_embed)
```

### Variables

```
FliProcessCbHdl *sim_init_cb
```

```
FliProcessCbHdl *sim_finish_cb
```

```
FliImpl *fli_table
```

### File FliImpl.h

### Functions

```
void cocotb_init ()
```

```
void handle_fli_callback (void *data)
```

```
class FliProcessCbHdl : public virtual GpiCbHdl
```

Subclassed by *FliShutdownCbHdl*, *FliSignalCbHdl*, *FliSimPhaseCbHdl*, *FliStartupCbHdl*, *FliTimedCbHdl*

### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

```
int cleanup_callback ()
```

## Public Functions

```
FliProcessCbHdl (GpiImplInterface *impl)
```

```
virtual ~FliProcessCbHdl ()
```

```
virtual int arm_callback () = 0
```

## Protected Attributes

```
mtiProcessIdT m_proc_hdl
```

```
bool m_sensitised
```

```
class FliSignalCbHdl : public FliProcessCbHdl, public GpiValueCbHdl
```

## cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

```
FliSignalCbHdl (GpiImplInterface *impl, FliSignalObjHdl *sig_hdl, unsigned int edge)
```

```
int arm_callback ()
```

## Public Functions

```
virtual ~FliSignalCbHdl ()
```

```
int cleanup_callback ()
```

## Private Members

```
mtiSignalIdT m_sig_hdl
```

```
class FliSimPhaseCbHdl : public FliProcessCbHdl
```

Subclassed by *FliNextPhaseCbHdl*, *FliReadOnlyCbHdl*, *FliReadWriteCbHdl*

## cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

```
int arm_callback ()
```

## Public Functions

**FliSimPhaseCbHdl** (*GpiImplInterface* \*impl, mtiProcessPriorityT priority)

**virtual** ~FliSimPhaseCbHdl ()

## Protected Attributes

mtiProcessPriorityT m\_priority

**class** FliReadWriteCbHdl : public *FliSimPhaseCbHdl*

## Public Functions

**FliReadWriteCbHdl** (*GpiImplInterface* \*impl)

**virtual** ~FliReadWriteCbHdl ()

**class** FliNextPhaseCbHdl : public *FliSimPhaseCbHdl*

## Public Functions

**FliNextPhaseCbHdl** (*GpiImplInterface* \*impl)

**virtual** ~FliNextPhaseCbHdl ()

**class** FliReadOnlyCbHdl : public *FliSimPhaseCbHdl*

## Public Functions

**FliReadOnlyCbHdl** (*GpiImplInterface* \*impl)

**virtual** ~FliReadOnlyCbHdl ()

**class** FliStartupCbHdl : public *FliProcessCbHdl*

## cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized. . .

int **arm\_callback** ()

int **run\_callback** ()

## Public Functions

**FliStartupCbHdl** (*GpiImplInterface* \*impl)

**virtual** ~FliStartupCbHdl ()

**class** FliShutdownCbHdl : public *FliProcessCbHdl*

### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

```
int arm_callback ()
```

```
int run_callback ()
```

### Public Functions

```
FliShutdownCbHdl (GpiImplInterface *impl)
```

```
virtual ~FliShutdownCbHdl ()
```

```
class FliTimedCbHdl : public FliProcessCbHdl
```

### cleanup callback

Called while unwinding after a GPI callback

We keep the process but desensitize it

NB: need a way to determine if should leave it sensitized...

```
FliTimedCbHdl (GpiImplInterface *impl, uint64_t time_ps)
```

```
int arm_callback ()
```

```
int cleanup_callback ()
```

### Public Functions

```
virtual ~FliTimedCbHdl ()
```

```
void reset_time (uint64_t new_time)
```

### Private Members

```
uint64_t m_time_ps
```

```
class FliObj
```

Subclassed by *FliObjHdl*, *FliSignalObjHdl*

### Public Functions

```
FliObj (int acc_type, int acc_full_type)
```

```
virtual ~FliObj ()
```

```
int get_acc_type ()
```

```
int get_acc_full_type ()
```



### Protected Attributes

```
int m_acc_type
int m_acc_full_type
```

```
class FliObjHdl : public GpiObjHdl, public FliObj
```

### Public Functions

```
FliObjHdl (GpiImplInterface *impl, void *hdl, gpi_objtype_t objtype, int acc_type, int acc_full_type)
```

```
FliObjHdl (GpiImplInterface *impl, void *hdl, gpi_objtype_t objtype, int acc_type, int acc_full_type,
            bool is_const)
```

```
virtual ~FliObjHdl ()
```

```
int initialise (std::string &name, std::string &fq_name)
```

```
class FliSignalObjHdl : public GpiSignalObjHdl, public FliObj
    Subclassed by FliValueObjHdl
```

### Public Functions

```
FliSignalObjHdl (GpiImplInterface *impl, void *hdl, gpi_objtype_t objtype, bool is_const, int
                  acc_type, int acc_full_type, bool is_var)
```

```
virtual ~FliSignalObjHdl ()
```

```
GpiCbHdl *value_change_cb (unsigned int edge)
```

```
int initialise (std::string &name, std::string &fq_name)
```

```
bool is_var ()
```

### Protected Attributes

```
bool m_is_var
FliSignalCbHdl m_rising_cb
FliSignalCbHdl m_falling_cb
FliSignalCbHdl m_either_cb
```

```
class FliValueObjHdl : public FliSignalObjHdl
    Subclassed by FliEnumObjHdl, FliIntObjHdl, FliLogicObjHdl, FliRealObjHdl, FliStringObjHdl
```

### Public Functions

```
FliValueObjHdl (GpiImplInterface *impl, void *hdl, gpi_objtype_t objtype, bool is_const, int
                  acc_type, int acc_full_type, bool is_var, mtiTypeIdT valType, mtiTypeKindT type-
                  Kind)
```

```
virtual ~FliValueObjHdl ()
```

```
const char *get_signal_value_binstr ()
```

```
const char *get_signal_value_str ()  
  
double get_signal_value_real ()  
  
long get_signal_value_long ()  
  
int set_signal_value (const long value)  
  
int set_signal_value (const double value)  
  
int set_signal_value (std::string &value)  
  
void *get_sub_hdl (int index)  
  
int initialise (std::string &name, std::string &fq_name)  
  
mtiTypeKindT get_fli_typekind ()  
  
mtiTypeIdT get_fli_typeid ()
```

### Protected Attributes

```
mtiTypeKindT m_fli_type  
mtiTypeIdT m_val_type  
char *m_val_buff  
void **m_sub_hdls  
  
class FliEnumObjHdl : public FliValueObjHdl
```

### Public Functions

```
FliEnumObjHdl (GpiImplInterface *impl, void *hdl, gpi_objtype_t objtype, bool is_const, int  
               acc_type, int acc_full_type, bool is_var, mtiTypeIdT valType, mtiTypeKindT type-  
               Kind)  
  
virtual ~FliEnumObjHdl ()  
  
const char *get_signal_value_str ()  
  
long get_signal_value_long ()  
  
int set_signal_value (const long value)  
  
int initialise (std::string &name, std::string &fq_name)
```

### Private Members

```
char **m_value_enum  
mtiInt32T m_num_enum  
  
class FliLogicObjHdl : public FliValueObjHdl
```

## Public Functions

**FliLogicObjHdl** (*GpiImplInterface* \*impl, void \*hdl, gpi\_objtype\_t objtype, bool is\_const, int acc\_type, int acc\_full\_type, bool is\_var, mtiTypeIdT valType, mtiTypeKindT typeKind)

**virtual** ~FliLogicObjHdl ()

**const** char \*get\_signal\_value\_binstr ()

int set\_signal\_value (const long value)

int set\_signal\_value (std::string &value)

int initialise (std::string &name, std::string &fq\_name)

## Private Members

char \*m\_mti\_buff

char \*\*m\_value\_enum

mtiInt32T m\_num\_enum

std::map<char, mtiInt32T> m\_enum\_map

**class** FliIntObjHdl : public *FliValueObjHdl*

## Public Functions

**FliIntObjHdl** (*GpiImplInterface* \*impl, void \*hdl, gpi\_objtype\_t objtype, bool is\_const, int acc\_type, int acc\_full\_type, bool is\_var, mtiTypeIdT valType, mtiTypeKindT typeKind)

**virtual** ~FliIntObjHdl ()

**const** char \*get\_signal\_value\_binstr ()

long get\_signal\_value\_long ()

int set\_signal\_value (const long value)

int initialise (std::string &name, std::string &fq\_name)

**class** FliRealObjHdl : public *FliValueObjHdl*

## Public Functions

**FliRealObjHdl** (*GpiImplInterface* \*impl, void \*hdl, gpi\_objtype\_t objtype, bool is\_const, int acc\_type, int acc\_full\_type, bool is\_var, mtiTypeIdT valType, mtiTypeKindT typeKind)

**virtual** ~FliRealObjHdl ()

double get\_signal\_value\_real ()

int set\_signal\_value (const double value)

int initialise (std::string &name, std::string &fq\_name)

### Private Members

double **\*m\_mti\_buff**

**class FliStringObjHdl** : public *FliValueObjHdl*

### Public Functions

**FliStringObjHdl** (*GpiImplInterface* \*impl, void \*hdl, gpi\_objtype\_t objtype, bool is\_const, int acc\_type, int acc\_full\_type, bool is\_var, mtiTypeIdT valType, mtiTypeKindT typeKind)

**virtual ~FliStringObjHdl** ()

**const** char \***get\_signal\_value\_str** ()

int **set\_signal\_value** (std::string &value)

int **initialise** (std::string &name, std::string &fq\_name)

### Private Members

char **\*m\_mti\_buff**

**class FliTimerCache**

### Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

*FliTimedCbHdl* \***get\_timer** (uint64\_t time\_ps)

void **put\_timer** (*FliTimedCbHdl* \*hdl)

### Public Functions

**FliTimerCache** (*FliImpl* \*impl)

**~FliTimerCache** ()

### Private Members

std::queue<*FliTimedCbHdl* \*> **free\_list**

*FliImpl* \*impl

**class FliIterator** : public *GpiIterator*

## Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

*GpiIteratorMapping*<int, *FliIterator*::OneToMany> **iterate\_over**

**FliIterator** (*GpiImplInterface* \*impl, *GpiObjHdl* \*hdl)

*GpiIterator*::Status **next\_handle** (std::string &name, *GpiObjHdl* \*\*hdl, void \*\*raw\_hdl)

void **populate\_handle\_list** (*OneToMany* childType)

## Public Types

**enum OneToMany**

Values:

**OTM\_END** = 0

**OTM\_CONSTANTS**

**OTM\_SIGNALS**

**OTM\_REGIONS**

**OTM\_SIGNAL\_SUB\_ELEMENTS**

**OTM\_VARIABLE\_SUB\_ELEMENTS**

## Public Functions

**virtual ~FliIterator** ()

## Private Members

std::vector<*OneToMany*> \*selected

std::vector<*OneToMany*>::iterator one2many

std::vector<void \*> m\_vars

std::vector<void \*> m\_sigs

std::vector<void \*> m\_regs

std::vector<void \*> \*m\_currentHandles

std::vector<void \*>::iterator m\_iterator

**class FliImpl** : public *GpiImplInterface*

## Native Check Create

Determine whether a simulation object is native to FLI and create a handle if it is

```
GpiObjHdl *native_check_create (std::string &name, GpiObjHdl *parent)
```

```
GpiObjHdl *native_check_create (int32_t index, GpiObjHdl *parent)
```

```
const char *reason_to_string (int reason)
```

## Get current simulation time

Get current simulation time

NB units depend on the simulation configuration

```
void get_sim_time (uint32_t *high, uint32_t *low)
```

```
void get_sim_precision (int32_t *precision)
```

## Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

```
GpiObjHdl *get_root_handle (const char *name)
```

```
Gpiterator *iterate_handle (GpiObjHdl *obj_hdl, gpi_iterator_sel_t type)
```

```
GpiCbHdl *register_timed_callback (uint64_t time_ps)
```

```
GpiCbHdl *register_readonly_callback ()
```

```
GpiCbHdl *register_nexttime_callback ()
```

```
GpiCbHdl *register_readwrite_callback ()
```

```
int deregister_callback (GpiCbHdl *obj_hdl)
```

## Public Functions

```
FliImpl (const std::string &name)
```

```
void sim_end ()
```

```
GpiObjHdl *native_check_create (void *raw_hdl, GpiObjHdl *parent)
```

```
GpiObjHdl *create_gpi_obj_from_handle (void *hdl, std::string &name, std::string &fq_name,  
int accType, int accFullType)
```

## Public Members

*FliTimerCache* **cache**

## Private Functions

bool **isValueConst** (int *kind*)  
bool **isValueLogic** (mtiTypeIdT *type*)  
bool **isValueChar** (mtiTypeIdT *type*)  
bool **isValueBoolean** (mtiTypeIdT *type*)  
bool **isTypeValue** (int *type*)  
bool **isTypeSignal** (int *type*, int *full\_type*)

## Private Members

*FliReadOnlyCbHdl* **m\_readonly\_cbhdl**  
*FliNextPhaseCbHdl* **m\_nexttime\_cbhdl**  
*FliReadWriteCbHdl* **m\_readwrite\_cbhdl**

## File FliObjHdl.cpp

## File GpiCbHdl.cpp

## Defines

```
ret = #_X; \ break ]
```

## File GpiCommon.cpp

## Defines

```
CHECK_AND_STORE (_x) _x  
DOT_LIB_EXT “.” xstr(LIB_EXT)
```

## Functions

```
int gpi_print_registered_impl ()  
int gpi_register_impl (GpiImplInterface *func_tbl)  
void gpi_embed_init (gpi_sim_info_t *info)  
void gpi_embed_end ()  
void gpi_sim_end ()  
void gpi_embed_event (gpi_event_t level, const char *msg)
```

```
static void gpi_load_libs (std::vector<std::string> to_load)
void gpi_load_extra_libs ()
void gpi_get_sim_time (uint32_t *high, uint32_t *low)
void gpi_get_sim_precision (int32_t *precision)
gpi_sim_hdl gpi_get_root_handle (const char *name)
static GpiObjHdl *__gpi_get_handle_by_name (GpiObjHdl *parent, std::string name, GpiImplInterface *skip_impl)
static GpiObjHdl *__gpi_get_handle_by_raw (GpiObjHdl *parent, void *raw_hdl, GpiImplInterface *skip_impl)
gpi_sim_hdl gpi_get_handle_by_name (gpi_sim_hdl parent, const char *name)
gpi_sim_hdl gpi_get_handle_by_index (gpi_sim_hdl parent, int32_t index)
gpi_iterator_hdl gpi_iterate (gpi_sim_hdl base, gpi_iterator_sel_t type)
gpi_sim_hdl gpi_next (gpi_iterator_hdl iterator)
const char *gpi_get_definition_name (gpi_sim_hdl sig_hdl)
const char *gpi_get_definition_file (gpi_sim_hdl sig_hdl)
const char *gpi_get_signal_value_binstr (gpi_sim_hdl sig_hdl)
const char *gpi_get_signal_value_str (gpi_sim_hdl sig_hdl)
double gpi_get_signal_value_real (gpi_sim_hdl sig_hdl)
long gpi_get_signal_value_long (gpi_sim_hdl sig_hdl)
const char *gpi_get_signal_name_str (gpi_sim_hdl sig_hdl)
const char *gpi_get_signal_type_str (gpi_sim_hdl sig_hdl)
gpi_objtype_t gpi_get_object_type (gpi_sim_hdl sig_hdl)
int gpi_is_constant (gpi_sim_hdl sig_hdl)
int gpi_is_indexable (gpi_sim_hdl sig_hdl)
void gpi_set_signal_value_long (gpi_sim_hdl sig_hdl, long value)
void gpi_set_signal_value_str (gpi_sim_hdl sig_hdl, const char *str)
void gpi_set_signal_value_real (gpi_sim_hdl sig_hdl, double value)
int gpi_get_num_elems (gpi_sim_hdl sig_hdl)
int gpi_get_range_left (gpi_sim_hdl sig_hdl)
int gpi_get_range_right (gpi_sim_hdl sig_hdl)
gpi_sim_hdl gpi_register_value_change_callback (int (*gpi_function)) const void *
    , void *gpi_cb_data, gpi_sim_hdl sig_hdl, unsigned int edge
gpi_sim_hdl gpi_register_timed_callback (int (*gpi_function)) const void *
    , void *gpi_cb_data, uint64_t time_ps
gpi_sim_hdl gpi_register_readonly_callback (int (*gpi_function)) const void *
    , void *gpi_cb_data
gpi_sim_hdl gpi_register_nexttime_callback (int (*gpi_function)) const void *
    , void *gpi_cb_data
```



```
gpi_sim_hdl gpi_register_readwrite_callback (int (*gpi_function)) const void *  
    , void *gpi_cb_data  
gpi_sim_hdl gpi_create_clock (gpi_sim_hdl clk_signal, const int period)  
void gpi_stop_clock (gpi_sim_hdl clk_object)  
void gpi_deregister_callback (gpi_sim_hdl hdl)
```

## Variables

```
vector<GpiImplInterface *> registered_impls
```

## File VhpiCbHdl.cpp

### Defines

```
VHPI_TYPE_MIN (1000)
```

### Functions

```
void handle_vhpi_callback (const vhpiCbDataT *cb_data)  
bool get_range (vhpiHandleT hdl, vhpiIntT dim, int *left, int *right)  
void vhpi_mappings (GpiIteratorMapping<vhpiClassKindT, vhpiOneToManyT> &map)
```

## File VhpiImpl.cpp

### Defines

```
CASE_STR (_X) case _X: return #_X
```

### Functions

```
bool is_const (vhpiHandleT hdl)  
bool is_enum_logic (vhpiHandleT hdl)  
bool is_enum_char (vhpiHandleT hdl)  
bool is_enum_boolean (vhpiHandleT hdl)  
void handle_vhpi_callback (const vhpiCbDataT *cb_data)  
static void register_initial_callback ()  
static void register_final_callback ()  
static void register_embed ()  
void vhpi_startup_routines_bootstrap ()
```

## Variables

```
VhpiCbHdl *sim_init_cb  
VhpiCbHdl *sim_finish_cb  
VhpiImpl *vhpi_table  
void (*vhpi_startup_routines[])() = {register_embed, , , }
```

## File VhpiImpl.h

### Defines

```
GEN_IDX_SEP_LHS “__”  
GEN_IDX_SEP_RHS “”  
__check_vhpi_error(__FILE__, __func__, __LINE__); \ } while (0) ]
```

### Functions

```
static int __check_vhpi_error (const char *file, const char *func, long line)  
class VhpiCbHdl : public virtual GpiCbHdl  
    Subclassed by VhpiNextPhaseCbHdl, VhpiReadOnlyCbHdl, VhpiReadwriteCbHdl, VhpiShutdownCbHdl, Vh-  
    piStartupCbHdl, VhpiTimedCbHdl, VhpiValueCbHdl
```

#### Public Functions

```
VhpiCbHdl (GpiImplInterface *impl)  
virtual ~VhpiCbHdl ()  
int arm_callback ()  
int cleanup_callback ()
```

#### Protected Attributes

```
vhpiCbDataT cb_data  
vhpiTimeT vhpi_time  
class VhpiValueCbHdl : public VhpiCbHdl, public GpiValueCbHdl
```

#### Public Functions

```
VhpiValueCbHdl (GpiImplInterface *impl, VhpiSignalObjHdl *sig, int edge)  
virtual ~VhpiValueCbHdl ()  
int cleanup_callback ()
```

### Private Members

std::string **initial\_value**

bool **rising**

bool **falling**

*VhpiSignalObjHdl* \***signal**

**class** VhpiTimedCbHdl : public *VhpiCbHdl*

### Public Functions

VhpiTimedCbHdl (*GpiImplInterface* \*impl, uint64\_t time\_ps)

virtual ~VhpiTimedCbHdl ()

int cleanup\_callback ()

**class** VhpiReadOnlyCbHdl : public *VhpiCbHdl*

### Public Functions

VhpiReadOnlyCbHdl (*GpiImplInterface* \*impl)

virtual ~VhpiReadOnlyCbHdl ()

**class** VhpiNextPhaseCbHdl : public *VhpiCbHdl*

### Public Functions

VhpiNextPhaseCbHdl (*GpiImplInterface* \*impl)

virtual ~VhpiNextPhaseCbHdl ()

**class** VhpiStartupCbHdl : public *VhpiCbHdl*

### Public Functions

VhpiStartupCbHdl (*GpiImplInterface* \*impl)

int run\_callback ()

int cleanup\_callback ()

virtual ~VhpiStartupCbHdl ()

**class** VhpiShutdownCbHdl : public *VhpiCbHdl*

### Public Functions

```
VhpiShutdownCbHdl (GpiImplInterface *impl)
int run_callback ()
int cleanup_callback ()
virtual ~VhpiShutdownCbHdl ()
class VhpiReadwriteCbHdl : public VhpiCbHdl
```

### Public Functions

```
VhpiReadwriteCbHdl (GpiImplInterface *impl)
virtual ~VhpiReadwriteCbHdl ()
class VhpiArrayObjHdl : public GpiObjHdl
```

### Public Functions

```
VhpiArrayObjHdl (GpiImplInterface *impl, vhpiHandleT hdl, gpi_objtype_t objtype)
virtual ~VhpiArrayObjHdl ()
int initialise (std::string &name, std::string &fq_name)
class VhpiObjHdl : public GpiObjHdl
```

### Public Functions

```
VhpiObjHdl (GpiImplInterface *impl, vhpiHandleT hdl, gpi_objtype_t objtype)
virtual ~VhpiObjHdl ()
int initialise (std::string &name, std::string &fq_name)
class VhpiSignalObjHdl : public GpiSignalObjHdl
Subclassed by VhpiLogicSignalObjHdl
```

### Public Functions

```
VhpiSignalObjHdl (GpiImplInterface *impl, vhpiHandleT hdl, gpi_objtype_t objtype, bool is_const)
~VhpiSignalObjHdl ()
const char *get_signal_value_binstr ()
const char *get_signal_value_str ()
double get_signal_value_real ()
long get_signal_value_long ()
int set_signal_value (const long value)
```

```

int set_signal_value (const double value)

int set_signal_value (std::string &value)

GpiCbHdl *value_change_cb (unsigned int edge)

int initialise (std::string &name, std::string &fq_name)

```

### Protected Functions

```

const vypiEnumT chr2vypi (const char value)

```

### Protected Attributes

```

vypiValueT m_value
vypiValueT m_binvalue
VypiValueCbHdl m_rising_cb
VypiValueCbHdl m_falling_cb
VypiValueCbHdl m_either_cb
class VypiLogicSignalObjHdl : public VypiSignalObjHdl

```

### Public Functions

```

VypiLogicSignalObjHdl (GpiImplInterface *impl, vypiHandleT hdl, gpi_objtype_t objtype, bool
                        is_const)

virtual ~VypiLogicSignalObjHdl ()

int set_signal_value (const long value)

int set_signal_value (std::string &value)

int initialise (std::string &name, std::string &fq_name)

class VypiIterator : public GpiIterator

```

### Public Functions

```

VypiIterator (GpiImplInterface *impl, GpiObjHdl *hdl)

~VypiIterator ()

GpiIterator::Status next_handle (std::string &name, GpiObjHdl **hdl, void **raw_hdl)

```

### Private Members

```

vypiHandleT m_iterator
vypiHandleT m_iter_obj
std::vector<vypiOneToManyT> *selected
std::vector<vypiOneToManyT>::iterator one2many

```

## Private Static Attributes

*GpiIteratorMapping*<vhpiClassKindT, vhpiOneToManyT> **iterate\_over**  
**class** *VhpiImpl* : **public** *GpiImplInterface*

## Public Functions

**VhpiImpl** (**const** std::string &*name*)  
**void** **sim\_end** ()  
**void** **get\_sim\_time** (uint32\_t \**high*, uint32\_t \**low*)  
**void** **get\_sim\_precision** (int32\_t \**precision*)  
*GpiObjHdl* \***get\_root\_handle** (**const** char \**name*)  
*GpiIterator* \***iterate\_handle** (*GpiObjHdl* \**obj\_hdl*, gpi\_iterator\_sel\_t *type*)  
*GpiCbHdl* \***register\_timed\_callback** (uint64\_t *time\_ps*)  
*GpiCbHdl* \***register\_readonly\_callback** ()  
*GpiCbHdl* \***register\_nexttime\_callback** ()  
*GpiCbHdl* \***register\_readwrite\_callback** ()  
**int** **deregister\_callback** (*GpiCbHdl* \**obj\_hdl*)  
*GpiObjHdl* \***native\_check\_create** (std::string &*name*, *GpiObjHdl* \**parent*)  
*GpiObjHdl* \***native\_check\_create** (int32\_t *index*, *GpiObjHdl* \**parent*)  
*GpiObjHdl* \***native\_check\_create** (void \**raw\_hdl*, *GpiObjHdl* \**parent*)  
**const** char \***reason\_to\_string** (int *reason*)  
**const** char \***format\_to\_string** (int *format*)  
*GpiObjHdl* \***create\_gpi\_obj\_from\_handle** (vhpiHandleT *new\_hdl*, std::string &*name*,  
std::string &*fq\_name*)

## Private Members

*VhpiReadwriteCbHdl* **m\_read\_write**  
*VhpiNextPhaseCbHdl* **m\_next\_phase**  
*VhpiReadOnlyCbHdl* **m\_read\_only**

## File VpiCbHdl.cpp

## Defines

**VPI\_TYPE\_MAX** (1000)

## Functions

```
int32_t handle_vpi_callback (p_cb_data cb_data)  
void vpi_mappings (GpiteratorMapping<int32_t, int32_t> &map)
```

## File VpiImpl.cpp

### Defines

```
CASE_STR (_X) case _X: return #_X
```

### Functions

```
gpi_objtype_t to_gpi_objtype (int32_t vpitype)  
int32_t handle_vpi_callback (p_cb_data cb_data)  
static void register_embed ()  
static void register_initial_callback ()  
static void register_final_callback ()  
static int system_function_compiletf (char *userdata)  
static int system_function_overload (char *userdata)  
static void register_system_functions ()  
void vlog_startup_routines_bootstrap ()
```

### Variables

```
VpiCbHdl *sim_init_cb  
VpiCbHdl *sim_finish_cb  
VpiImpl *vpi_table  
int systf_info_level = GPIInfo  
int systf_warning_level = GPIWarning  
int systf_error_level = GPIError  
int systf_fatal_level = GPICritical  
void (*vlog_startup_routines[])() = {register_embed, , , }
```

## File VpiImpl.h

### Defines

```
__check_vpi_error(__FILE__, __func__, __LINE__); \ } while (0) ]
```

## Functions

```
static int __check_vpi_error (const char *file, const char *func, long line)
```

```
class VpiCbHdl : public virtual GpiCbHdl
```

Subclassed by *VpiNextPhaseCbHdl*, *VpiReadOnlyCbHdl*, *VpiReadwriteCbHdl*, *VpiShutdownCbHdl*, *VpiStartupCbHdl*, *VpiTimedCbHdl*, *VpiValueCbHdl*

### Public Functions

```
VpiCbHdl (GpiImplInterface *impl)
```

```
virtual ~VpiCbHdl ()
```

```
int arm_callback ()
```

```
int cleanup_callback ()
```

### Protected Attributes

```
s_cb_data cb_data
```

```
s_vpi_time vpi_time
```

```
class VpiValueCbHdl : public VpiCbHdl, public GpiValueCbHdl
```

### Public Functions

```
VpiValueCbHdl (GpiImplInterface *impl, VpiSignalObjHdl *sig, int edge)
```

```
virtual ~VpiValueCbHdl ()
```

```
int cleanup_callback ()
```

### Private Members

```
s_vpi_value m_vpi_value
```

```
class VpiTimedCbHdl : public VpiCbHdl
```

### Public Functions

```
VpiTimedCbHdl (GpiImplInterface *impl, uint64_t time_ps)
```

```
virtual ~VpiTimedCbHdl ()
```

```
int cleanup_callback ()
```

```
class VpiReadOnlyCbHdl : public VpiCbHdl
```



### Public Functions

```
VpiReadOnlyCbHdl (GpiImplInterface *impl)
virtual ~VpiReadOnlyCbHdl ()
class VpiNextPhaseCbHdl : public VpiCbHdl
```

### Public Functions

```
VpiNextPhaseCbHdl (GpiImplInterface *impl)
virtual ~VpiNextPhaseCbHdl ()
class VpiReadwriteCbHdl : public VpiCbHdl
```

### Public Functions

```
VpiReadwriteCbHdl (GpiImplInterface *impl)
virtual ~VpiReadwriteCbHdl ()
class VpiStartupCbHdl : public VpiCbHdl
```

### Public Functions

```
VpiStartupCbHdl (GpiImplInterface *impl)
int run_callback ()
int cleanup_callback ()
virtual ~VpiStartupCbHdl ()
class VpiShutdownCbHdl : public VpiCbHdl
```

### Public Functions

```
VpiShutdownCbHdl (GpiImplInterface *impl)
int run_callback ()
int cleanup_callback ()
virtual ~VpiShutdownCbHdl ()
class VpiArrayObjHdl : public GpiObjHdl
```

### Public Functions

```
VpiArrayObjHdl (GpiImplInterface *impl, vpiHandle hdl, gpi_objtype_t objtype)
virtual ~VpiArrayObjHdl ()
int initialise (std::string &name, std::string &fq_name)
class VpiObjHdl : public GpiObjHdl
```

### Public Functions

```
VpiObjHdl (GpiImplInterface *impl, vpiHandle hdl, gpi_objtype_t objtype)
virtual ~VpiObjHdl ()
int initialise (std::string &name, std::string &fq_name)
class VpiSignalObjHdl : public GpiSignalObjHdl
```

### Public Functions

```
VpiSignalObjHdl (GpiImplInterface *impl, vpiHandle hdl, gpi_objtype_t objtype, bool is_const)
virtual ~VpiSignalObjHdl ()
const char *get_signal_value_binstr ()
const char *get_signal_value_str ()
double get_signal_value_real ()
long get_signal_value_long ()
int set_signal_value (const long value)
int set_signal_value (const double value)
int set_signal_value (std::string &value)
GpiCbHdl *value_change_cb (unsigned int edge)
int initialise (std::string &name, std::string &fq_name)
```

### Private Functions

```
int set_signal_value (s_vpi_value value)
```

### Private Members

```
VpiValueCbHdl m_rising_cb
VpiValueCbHdl m_falling_cb
VpiValueCbHdl m_either_cb
class VpiIterator : public GpiIterator
```

## Public Functions

**VpiIterator** (*GpiImplInterface* \*impl, *GpiObjHdl* \*hdl)

**~VpiIterator** ()

*GpiIterator*::Status **next\_handle** (std::string &name, *GpiObjHdl* \*\*hdl, void \*\*raw\_hdl)

## Private Members

vpiHandle **m\_iterator**

std::vector<int32\_t> \***selected**

std::vector<int32\_t>::iterator **one2many**

## Private Static Attributes

*GpiIteratorMapping*<int32\_t, int32\_t> **iterate\_over**

**class VpiSingleIterator** : public *GpiIterator*

## Public Functions

**VpiSingleIterator** (*GpiImplInterface* \*impl, *GpiObjHdl* \*hdl, int32\_t vptype)

**virtual ~VpiSingleIterator** ()

*GpiIterator*::Status **next\_handle** (std::string &name, *GpiObjHdl* \*\*hdl, void \*\*raw\_hdl)

## Protected Attributes

vpiHandle **m\_iterator**

**class VpiImpl** : public *GpiImplInterface*

## Public Functions

**VpiImpl** (const std::string &name)

void **sim\_end** ()

void **get\_sim\_time** (uint32\_t \*high, uint32\_t \*low)

void **get\_sim\_precision** (int32\_t \*precision)

*GpiObjHdl* \***get\_root\_handle** (const char \*name)

*GpiIterator* \***iterate\_handle** (*GpiObjHdl* \*obj\_hdl, gpi\_iterator\_sel\_t type)

*GpiObjHdl* \***next\_handle** (*GpiIterator* \*iter)

*GpiCbHdl* \***register\_timed\_callback** (uint64\_t time\_ps)

*GpiCbHdl* \***register\_readonly\_callback** ()

```
GpiCbHdl *register_nexttime_callback ()
GpiCbHdl *register_readwrite_callback ()
int deregister_callback (GpiCbHdl *obj_hdl)
GpiObjHdl *native_check_create (std::string &name, GpiObjHdl *parent)
GpiObjHdl *native_check_create (int32_t index, GpiObjHdl *parent)
GpiObjHdl *native_check_create (void *raw_hdl, GpiObjHdl *parent)
const char *reason_to_string (int reason)
GpiObjHdl *create_gpi_obj_from_handle (vpiHandle new_hdl, std::string &name, std::string
&fq_name)
```

### Private Members

```
VpiReadwriteCbHdl m_read_write
VpiNextPhaseCbHdl m_next_phase
VpiReadOnlyCbHdl m_read_only
```

## File cocotb\_utils.c

### Functions

```
void to_python (void)
void to_simulator (void)
void *utils_dyn_open (const char *lib_name)
void *utils_dyn_sym (void *handle, const char *sym_name)
```

### Variables

```
int is_python_context = 0
```

## File entrypoint.vhdl

```
class cocotb_entrypoint
```

```
    class cocotb_arch
```

### Public Members

```
    cocotb_arch:architecture is      "cocotb_fli_init fli.so" cocotb_entrypoint.cocotb_ar
```

## File gpi\_embed.c

### Initialization

Called by the simulator on initialization.

Load cocotb Python module

GILState before calling: Not held

GILState after calling: Not held

Makes one call to PyGILState\_Ensure and one call to PyGILState\_Release

Loads the Python module called cocotb and calls the `_initialise_testbench` function

**COCOTB\_MODULE** "cocotb"

```
int get_module_ref (const char *modname, PyObject **mod)
```

```
int embed_sim_init (gpi_sim_info_t *info)
```

```
void embed_sim_event (gpi_event_t level, const char *msg)
```

### Initialize the Python interpreter

Create and initialize the Python interpreter

GILState before calling: N/A

GILState after calling: released

Stores the thread state for cocotb in static variable `gtstate`

```
void embed_init_python (void)
```

### Functions

```
static void set_program_name_in_venv (void)
```

### Variables

```
PyThreadState *gtstate = NULL
```

```
char progrname[] = "cocotb"
```

```
char *argv[] = {progrname}
```

```
const char *PYTHON_INTERPRETER_PATH = "/bin/python"
```

```
PyObject *pEventFn = NULL
```

## File gpi\_logging.c

### GPI logging

Write a log message using cocotb SimLog class

GILState before calling: Unknown

GILState after calling: Unknown

Makes one call to `PyGILState_Ensure` and one call to `PyGILState_Release`

If the Python logging mechanism is not initialised, dumps to `stderr`.

```
void gpi_log (const char *name, long level, const char *pathname, const char *funcname, long lineno,  
             const char *msg, ...)
```

## Defines

```
LOG_SIZE 512
```

## Functions

```
void set_log_handler (void *handler)  
void set_log_filter (void *filter)  
void set_log_level (enum gpi_log_levels new_level)  
const char *log_level (long level)
```

## Variables

```
PyObject *pLogHandler  
PyObject *pLogFilter  
gpi_log_levels local_level = GPIInfo  
struct _log_level_table log_level_table[] = { { 10, "DEBUG" }, { 20,  
char log_buff[LOG_SIZE]  
struct _log_level_table
```

## Public Members

```
long level  
const char *levelname
```

## File gpi\_priv.h

## Defines

```
const void NAME##_entry_point() { \ func(); \ } }
```

## Typedefs

```
typedef enum gpi_cb_state gpi_cb_state_e  
typedef const void (*layer_entry_func)()
```

## Enums

enum **gpi\_cb\_state**

*Values:*

**GPI\_FREE** = 0

**GPI\_PRIMED** = 1

**GPI\_CALL** = 2

**GPI\_REPRIME** = 3

**GPI\_DELETE** = 4

## Functions

template<class **To**>

*To* **sim\_to\_hdl** (*gpi\_sim\_hdl input*)

int **gpi\_register\_impl** (*GpiImplInterface \*func\_tbl*)

void **gpi\_embed\_init** (*gpi\_sim\_info\_t \*info*)

void **gpi\_embed\_end** ()

void **gpi\_embed\_event** (*gpi\_event\_t level, const char \*msg*)

void **gpi\_load\_extra\_libs** ()

class **GpiHdl**

Subclassed by *GpiCbHdl*, *GpiIterator*, *GpiObjHdl*

### Public Functions

**GpiHdl** (*GpiImplInterface \*impl*)

**GpiHdl** (*GpiImplInterface \*impl, void \*hdl*)

**virtual ~GpiHdl** ()

int **initialise** (*std::string &name*)

template<typename **T**>

*T* **get\_handle** () **const**

char \***gpi\_copy\_name** (*const char \*name*)

bool **is\_this\_impl** (*GpiImplInterface \*impl*)

### Public Members

*GpiImplInterface* \***m\_impl**

### Protected Attributes

void \***m\_obj\_hdl**

## Private Functions

**GpiHdl** ()

**class** **GpiObjHdl** : **public** *GpiHdl*

Subclassed by *FliObjHdl*, *GpiSignalObjHdl*, *VhpiArrayObjHdl*, *VhpiObjHdl*, *VpiArrayObjHdl*, *VpiObjHdl*

## Public Functions

**GpiObjHdl** (*GpiImplInterface* \*impl)

**GpiObjHdl** (*GpiImplInterface* \*impl, void \*hdl, gpi\_objtype\_t objtype)

**GpiObjHdl** (*GpiImplInterface* \*impl, void \*hdl, gpi\_objtype\_t objtype, bool is\_const)

**virtual** ~**GpiObjHdl** ()

**const** char \***get\_name\_str** ()

**const** char \***get\_fullname\_str** ()

**const** char \***get\_type\_str** ()

gpi\_objtype\_t **get\_type** ()

bool **get\_const** ()

int **get\_num\_elems** ()

int **get\_range\_left** ()

int **get\_range\_right** ()

int **get\_indexable** ()

**const** std::string &**get\_name** ()

**const** std::string &**get\_fullname** ()

**virtual const** char \***get\_definition\_name** ()

**virtual const** char \***get\_definition\_file** ()

bool **is\_native\_impl** (*GpiImplInterface* \*impl)

int **initialise** (std::string &name, std::string &full\_name)

## Protected Attributes

int **m\_num\_elems**

bool **m\_indexable**

int **m\_range\_left**

int **m\_range\_right**

std::string **m\_name**

std::string **m\_fullname**



```
std::string m_definition_name
std::string m_definition_file
gpi_objtype_t m_type
bool m_const
```

```
class GpiSignalObjHdl : public GpiObjHdl
    Subclassed by FliSignalObjHdl, VhpiSignalObjHdl, VpiSignalObjHdl
```

### Public Functions

```
GpiSignalObjHdl (GpiImplInterface *impl, void *hdl, gpi_objtype_t objtype, bool is_const)

virtual ~GpiSignalObjHdl ()

virtual const char *get_signal_value_binstr () = 0
virtual const char *get_signal_value_str () = 0
virtual double get_signal_value_real () = 0
virtual long get_signal_value_long () = 0
virtual int set_signal_value (const long value) = 0
virtual int set_signal_value (const double value) = 0
virtual int set_signal_value (std::string &value) = 0
virtual GpiCbHdl *value_change_cb (unsigned int edge) = 0
```

### Public Members

```
int m_length
```

```
class GpiCbHdl : public GpiHdl
    Subclassed by FliProcessCbHdl, GpiValueCbHdl, VhpiCbHdl, VpiCbHdl
```

### Public Functions

```
GpiCbHdl (GpiImplInterface *impl)

int arm_callback () = 0
int run_callback ()
int cleanup_callback () = 0
int set_user_data (int (*gpi_function)) const void *
    , const void *data
const void *get_user_data ()
void set_call_state (gpi_cb_state_e new_state)
gpi_cb_state_e get_call_state ()

~GpiCbHdl ()
```

### Protected Attributes

```
int (*gpi_function) (const void *)
```

```
const void *m_cb_data
```

```
gpi_cb_state_e m_state
```

```
class GpiValueCbHdl : public virtual GpiCbHdl
```

```
Subclassed by FliSignalCbHdl, VhpiValueCbHdl, VpiValueCbHdl
```

### Public Functions

```
GpiValueCbHdl (GpiImplInterface *impl, GpiSignalObjHdl *signal, int edge)
```

```
virtual ~GpiValueCbHdl ()
```

```
int run_callback ()
```

```
virtual int cleanup_callback () = 0
```

### Protected Attributes

```
std::string required_value
```

```
GpiSignalObjHdl *m_signal
```

```
class GpiClockHdl
```

### Public Functions

```
GpiClockHdl (GpiObjHdl *clk)
```

```
GpiClockHdl (const char *clk)
```

```
~GpiClockHdl ()
```

```
int start_clock (const int period_ps)
```

```
int stop_clock ()
```

```
class GpiIterator : public GpiHdl
```

```
Subclassed by FliIterator, VhpiIterator, VpiIterator, VpiSingleIterator
```

### Public Types

```
enum Status
```

```
Values:
```

```
NATIVE
```

```
NATIVE_NO_NAME
```

```
NOT_NATIVE
```

```
NOT_NATIVE_NO_NAME
```

```
END
```

## Public Functions

```
GpiIterator (GpiImplInterface *impl, GpiObjHdl *hdl)
virtual ~GpiIterator ()
virtual Status next_handle (std::string &name, GpiObjHdl **hdl, void **raw_hdl)
GpiObjHdl *get_parent ()
```

## Protected Attributes

```
GpiObjHdl *m_parent
```

```
template<class Ti, class Tm>
class GpiIteratorMapping
```

## Public Functions

```
GpiIteratorMapping (void (*populate)) GpiIteratorMapping<Ti, Tm>&
std::vector<Tm> *get_options (Ti type)
void add_to_options (Ti type, Tm *options)
```

## Private Members

```
std::map<Ti, std::vector<Tm>> options_map
```

```
class GpiImplInterface
  Subclassed by FliImpl, VhpiImpl, VpiImpl
```

## Public Functions

```
GpiImplInterface (const std::string &name)
const char *get_name_c ()
const string &get_name_s ()
virtual ~GpiImplInterface ()
virtual void sim_end () = 0
virtual void get_sim_time (uint32_t *high, uint32_t *low) = 0
virtual void get_sim_precision (int32_t *precision) = 0
virtual GpiObjHdl *native_check_create (std::string &name, GpiObjHdl *parent) = 0
virtual GpiObjHdl *native_check_create (int32_t index, GpiObjHdl *parent) = 0
virtual GpiObjHdl *native_check_create (void *raw_hdl, GpiObjHdl *parent) = 0
virtual GpiObjHdl *get_root_handle (const char *name) = 0
```

```
virtual GpiIterator *iterate_handle (GpiObjHdl *obj_hdl, gpi_iterator_sel_t type) = 0  
virtual GpiCbHdl *register_timed_callback (uint64_t time_ps) = 0  
virtual GpiCbHdl *register_readonly_callback () = 0  
virtual GpiCbHdl *register_nexttime_callback () = 0  
virtual GpiCbHdl *register_readwrite_callback () = 0  
virtual int deregister_callback (GpiCbHdl *obj_hdl) = 0  
virtual const char *reason_to_string (int reason) = 0
```

### Private Members

```
std::string m_name
```

## File python3\_compat.h

### Defines

```
GETSTATE (m) (&_state)  
MODULE_ENTRY_POINT initsimulator  
INITERROR return  
struct module_state
```

### Public Members

```
PyObject *error
```

## File simulatormodule.c

Python extension to provide access to the simulator.

Uses GPI calls to interface to the simulator.

### Callback Handling

Handle a callback coming from GPI

GILState before calling: Unknown

GILState after calling: Unknown

Makes one call to TAKE\_GIL and one call to DROP\_GIL

Returns 0 on success or 1 on a failure.

Handles a callback from the simulator, all of which call this function.

We extract the associated context and find the Python function (usually cocotb.scheduler.react) calling it with a reference to the trigger that fired. The scheduler can then call next() on all the coroutines that are waiting on that particular trigger.

TODO:

- Tidy up return values
- Ensure cleanup correctly in exception cases

```
int handle_gpi_callback (void *user_data)

static PyObject *log_msg (PyObject *self, PyObject *args)

static PyObject *register_readonly_callback (PyObject *self, PyObject *args)

static PyObject *register_rwsynch_callback (PyObject *self, PyObject *args)

static PyObject *register_nextstep_callback (PyObject *self, PyObject *args)

static PyObject *register_timed_callback (PyObject *self, PyObject *args)

static PyObject *register_value_change_callback (PyObject *self, PyObject *args)

static PyObject *iterate (PyObject *self, PyObject *args)

static PyObject *next (PyObject *self, PyObject *args)

static PyObject *get_signal_val_binstr (PyObject *self, PyObject *args)

static PyObject *get_signal_val_str (PyObject *self, PyObject *args)

static PyObject *get_signal_val_real (PyObject *self, PyObject *args)

static PyObject *get_signal_val_long (PyObject *self, PyObject *args)

static PyObject *set_signal_val_str (PyObject *self, PyObject *args)

static PyObject *set_signal_val_real (PyObject *self, PyObject *args)

static PyObject *set_signal_val_long (PyObject *self, PyObject *args)

static PyObject *get_definition_name (PyObject *self, PyObject *args)

static PyObject *get_definition_file (PyObject *self, PyObject *args)

static PyObject *get_handle_by_name (PyObject *self, PyObject *args)

static PyObject *get_handle_by_index (PyObject *self, PyObject *args)

static PyObject *get_root_handle (PyObject *self, PyObject *args)

static PyObject *get_name_string (PyObject *self, PyObject *args)

static PyObject *get_type (PyObject *self, PyObject *args)

static PyObject *get_const (PyObject *self, PyObject *args)

static PyObject *get_type_string (PyObject *self, PyObject *args)

static PyObject *get_sim_time (PyObject *self, PyObject *args)

static PyObject *get_precision (PyObject *self, PyObject *args)

static PyObject *get_num_elems (PyObject *self, PyObject *args)

static PyObject *get_range (PyObject *self, PyObject *args)

static PyObject *stop_simulator (PyObject *self, PyObject *args)

static PyObject *deregister_callback (PyObject *self, PyObject *args)

static PyObject *log_level (PyObject *self, PyObject *args)

static void add_module_constants (PyObject *simulator)
```

## Typedefs

```
typedef int (*gpi_function_t) (const void *)
```

## Functions

```
PyGILState_STATE TAKE_GIL (void)
```

```
void DROP_GIL (PyGILState_STATE state)
```

```
static int gpi_sim_hdl_converter (PyObject *o, gpi_sim_hdl *data)
```

```
static int gpi_iterator_hdl_converter (PyObject *o, gpi_iterator_hdl *data)
```

## Variables

```
int takes = 0
```

```
int releases = 0
```

```
struct sim_time cache_time
```

```
struct sim_time
```

### Public Members

```
uint32_t high
```

```
uint32_t low
```

## File simulatormodule.h

### Defines

```
COCOTB_ACTIVE_ID 0xC0C07B
```

```
COCOTB_INACTIVE_ID 0xDEADB175
```

```
MODULE_NAME “simulator”
```

## Typedefs

```
typedef struct t_callback_data s_callback_data
```

```
typedef struct t_callback_data *p_callback_data
```

## Functions

```
static PyObject *error_out (PyObject *m)
```

```
static PyObject *log_msg (PyObject *self, PyObject *args)
```

```
static PyObject *get_signal_val_long (PyObject *self, PyObject *args)
```

```
static PyObject *get_signal_val_real (PyObject *self, PyObject *args)
```

```
static PyObject *get_signal_val_str (PyObject *self, PyObject *args)
```

```

static PyObject *get_signal_val_binstr (PyObject *self, PyObject *args)
static PyObject *set_signal_val_long (PyObject *self, PyObject *args)
static PyObject *set_signal_val_real (PyObject *self, PyObject *args)
static PyObject *set_signal_val_str (PyObject *self, PyObject *args)
static PyObject *get_definition_name (PyObject *self, PyObject *args)
static PyObject *get_definition_file (PyObject *self, PyObject *args)
static PyObject *get_handle_by_name (PyObject *self, PyObject *args)
static PyObject *get_handle_by_index (PyObject *self, PyObject *args)
static PyObject *get_root_handle (PyObject *self, PyObject *args)
static PyObject *get_name_string (PyObject *self, PyObject *args)
static PyObject *get_type (PyObject *self, PyObject *args)
static PyObject *get_const (PyObject *self, PyObject *args)
static PyObject *get_type_string (PyObject *self, PyObject *args)
static PyObject *get_num_elems (PyObject *self, PyObject *args)
static PyObject *get_range (PyObject *self, PyObject *args)
static PyObject *register_timed_callback (PyObject *self, PyObject *args)
static PyObject *register_value_change_callback (PyObject *self, PyObject *args)
static PyObject *register_readonly_callback (PyObject *self, PyObject *args)
static PyObject *register_nextstep_callback (PyObject *self, PyObject *args)
static PyObject *register_rwsynch_callback (PyObject *self, PyObject *args)
static PyObject *stop_simulator (PyObject *self, PyObject *args)
static PyObject *iterate (PyObject *self, PyObject *args)
static PyObject *next (PyObject *self, PyObject *args)
static PyObject *get_sim_time (PyObject *self, PyObject *args)
static PyObject *get_precision (PyObject *self, PyObject *args)
static PyObject *deregister_callback (PyObject *self, PyObject *args)
static PyObject *log_level (PyObject *self, PyObject *args)

```

## Variables

PyMethodDef **SimulatorMethods**[]

struct **t\_callback\_data**

## Public Members

PyThreadState \***\_saved\_thread\_state**

uint32\_t **id\_value**

PyObject \***function**

```
PyObject *args
PyObject *kwargs
gpi_sim_hdl cb_hdl
```

### File `simulatoremodule_python2.c`

#### Functions

```
static PyObject *error_out (PyObject *m)
PyMODINIT_FUNC MODULE_ENTRY_POINT (void)
```

#### Variables

```
char error_module[] = MODULE_NAME ".Error"
struct module_state _state
```

### File `simulatoremodule_python3.c`

#### Functions

```
static PyObject *error_out (PyObject *m)
static int simulator_traverse (PyObject *m, visitproc visit, void *arg)
static int simulator_clear (PyObject *m)
PyMODINIT_FUNC MODULE_ENTRY_POINT (void)
```

#### Variables

```
struct PyModuleDef moduledef = {PyModuleDef_HEAD_INIT, , , , , , }
```

## 8.1.3 Struct list

### Struct `_log_level_table`

```
struct _log_level_table
```

### Struct `module_state`

```
struct module_state
```

### Struct `sim_time`

```
struct sim_time
```



### Struct `t_callback_data`

```
struct t_callback_data
```



## TUTORIAL: ENDIAN SWAPPER

In this tutorial we'll use some of the built-in features of cocotb to quickly create a complex testbench.

---

**Note:** All the code and sample output from this example are available on [EDA Playground](#)

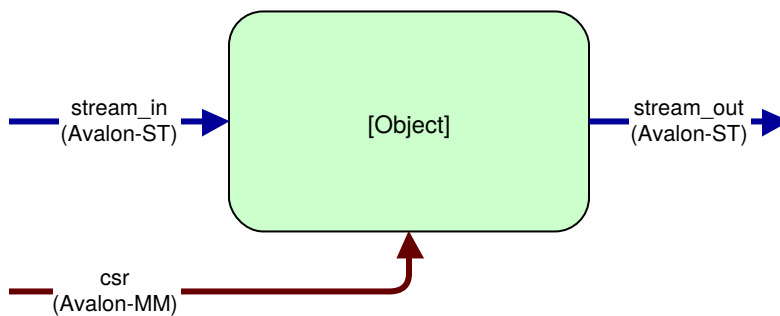
---

For the impatient this tutorial is provided as an example with cocotb. You can run this example from a fresh checkout:

```
cd examples/endian_swapper/tests
make
```

### 9.1 Design

We have a relatively simplistic RTL block called the `endian_swapper`. The DUT has three interfaces, all conforming to the Avalon standard:



The DUT will swap the endianness of packets on the Avalon-ST bus if a configuration bit is set. For every packet arriving on the `stream_in` interface the entire packet will be endian swapped if the configuration bit is set, otherwise the entire packet will pass through unmodified.

### 9.2 Testbench

To begin with we create a class to encapsulate all the common code for the testbench. It is possible to write directed tests without using a testbench class however to encourage code re-use it is good practice to create a distinct class.

```
class EndianSwapperTB(object):

    def __init__(self, dut):
```

(continues on next page)

(continued from previous page)

```

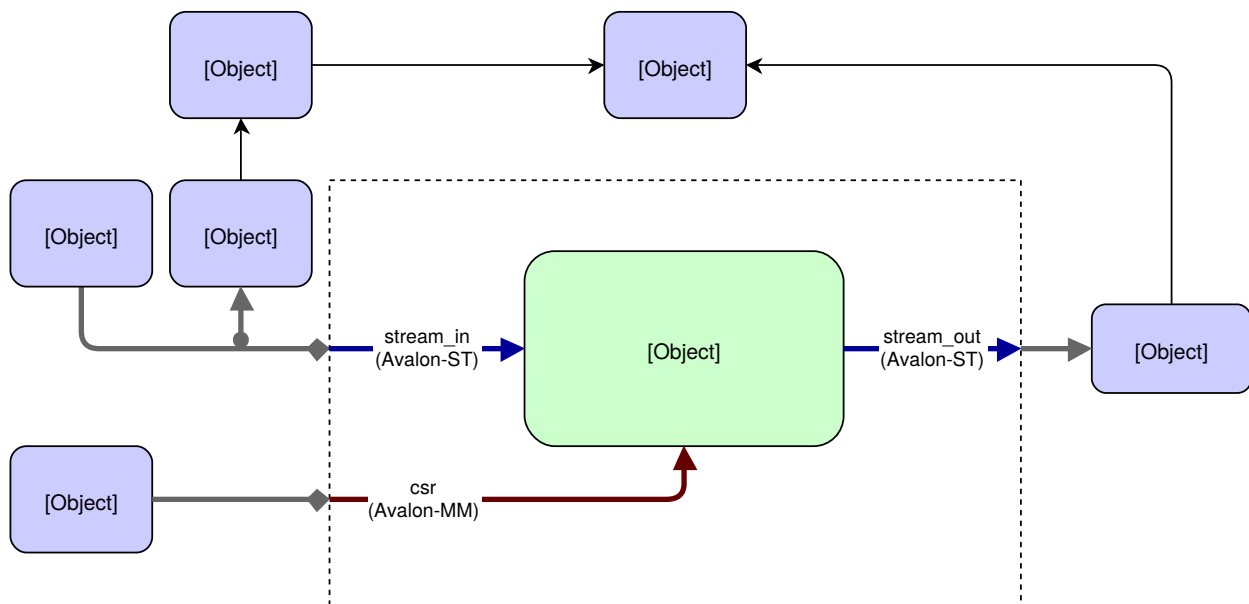
self.dut = dut
self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)
self.stream_out = AvalonSTMonitor(dut, "stream_out", dut.clk)
self.csr = AvalonMaster(dut, "csr", dut.clk)

self.expected_output = []
self.scoreboard = Scoreboard(dut)
self.scoreboard.add_interface(self.stream_out, self.expected_output)

# Reconstruct the input transactions from the pins and send them to our 'model'
↪ '
    self.stream_in_recovered = AvalonSTMonitor(dut, "stream_in", dut.clk, ↪
↪ callback=self.model)

```

With the above code we have created a testbench with the following structure:



If we inspect this line-by-line:

```
self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)
```

Here we are creating an `AvalonSTDriver` instance. The constructor requires 3 arguments - a handle to the entity containing the interface (`dut`), the name of the interface (`stream_in`) and the associated clock with which to drive the interface (`dut.clk`). The driver will auto-discover the signals for the interface, assuming that they follow the naming convention `<interface_name>_<signal>`.

In this case we have the following signals defined for the `stream_in` interface:

Name	Type	Description (from Avalon Specification)
<code>stream_in_data</code>	data	The data signal from the source to the sink
<code>stream_in_empty</code>	empty	Indicates the number of symbols that are empty during cycles that contain the end of a packet
<code>stream_in_valid</code>	valid	Asserted by the source to qualify all other source to sink signals
<code>stream_in_startofpacket</code>	startofpacket	Asserted by the source to mark the beginning of a packet
<code>stream_in_endofpacket</code>	endofpacket	Asserted by the source to mark the end of a packet
<code>stream_in_ready</code>	ready	Asserted high to indicate that the sink can accept data

By following the signal naming convention the driver can find the signals associated with this interface automatically.

```
self.stream_out = AvalonSTMonitor(dut, "stream_out", dut.clk)
self.csr = AvalonMaster(dut, "csr", dut.clk)
```

We do the same to create the *monitor* on *stream\_out* and the CSR interface.

```
self.expected_output = []
self.scoreboard = Scoreboard(dut)
self.scoreboard.add_interface(self.stream_out, self.expected_output)
```

The above lines create a *Scoreboard* instance and attach it to the *stream\_out* monitor instance. The scoreboard is used to check that the DUT behavior is correct. The call to *add\_interface()* takes a Monitor instance as the first argument and the second argument is a mechanism for describing the expected output for that interface. This could be a callable function but in this example a simple list of expected transactions is sufficient.

```
# Reconstruct the input transactions from the pins and send them to our 'model'
self.stream_in_recovered = AvalonSTMonitor(dut, "stream_in", dut.clk, callback=self.
    ↪model)
```

Finally we create another Monitor instance, this time connected to the *stream\_in* interface. This is to reconstruct the transactions being driven into the DUT. It's good practice to use a monitor to reconstruct the transactions from the pin interactions rather than snooping them from a higher abstraction layer as we can gain confidence that our drivers and monitors are functioning correctly.

We also pass the keyword argument *callback* to the monitor constructor which will result in the supplied function being called for each transaction seen on the bus with the transaction as the first argument. Our model function is quite straightforward in this case - we simply append the transaction to the expected output list and increment a counter:

```
def model(self, transaction):
    """Model the DUT based on the input transaction"""
    self.expected_output.append(transaction)
    self.pkts_sent += 1
```

## 9.2.1 Test Function

There are various ‘knobs’ we can tweak on this testbench to vary the behavior:

- Packet size
- Backpressure on the *stream\_out* interface
- Idle cycles on the *stream\_in* interface
- Configuration switching of the endian swap register during the test.

We want to run different variations of tests but they will all have a very similar structure so we create a common *run\_test* function. To generate backpressure on the *stream\_out* interface we use the *BitDriver* class from *cocotb.drivers*.

```
@cocotb.coroutine
def run_test(dut, data_in=None, config_coroutine=None, idle_inserter=None, ↪
    ↪backpressure_inserter=None):

    cocotb.fork(Clock(dut.clk, 5000).start())
    tb = EndianSwapperTB(dut)
```

(continues on next page)

(continued from previous page)

```

yield tb.reset()
dut.stream_out_ready <= 1

# Start off any optional coroutines
if config_coroutine is not None:
    cocotb.fork(config_coroutine(tb.csr))
if idle_inserter is not None:
    tb.stream_in.set_valid_generator(idle_inserter())
if backpressure_inserter is not None:
    tb.backpressure.start(backpressure_inserter())

# Send in the packets
for transaction in data_in():
    yield tb.stream_in.send(transaction)

# Wait at least 2 cycles where output ready is low before ending the test
for i in range(2):
    yield RisingEdge(dut.clk)
    while not dut.stream_out_ready.value:
        yield RisingEdge(dut.clk)

pkt_count = yield tb.csr.read(1)

if pkt_count.integer != tb.pkts_sent:
    raise TestFailure("DUT recorded %d packets but tb counted %d" % (
        pkt_count.integer, tb.pkts_sent))
else:
    dut._log.info("DUT correctly counted %d packets" % pkt_count.integer)

raise tb.scoreboard.result

```

We can see that this test function creates an instance of the testbench, resets the DUT by running the coroutine `tb.reset()` and then starts off any optional coroutines passed in using the keyword arguments. We then send in all the packets from `data_in`, ensure that all the packets have been received by waiting 2 cycles at the end. We read the packet count and compare this with the number of packets. Finally we use the `tb.scoreboard.result` to determine the status of the test. If any transactions didn't match the expected output then this member would be an instance of the `TestFailure` result.

## 9.2.2 Test permutations

Having defined a test function we can now auto-generate different permutations of tests using the `TestFactory` class:

```

factory = TestFactory(run_test)
factory.add_option("data_in", [random_packet_sizes])
factory.add_option("config_coroutine", [None, randomly_switch_config])
factory.add_option("idle_inserter", [None, wave, intermittent_single_cycles,
    ↪ random_50_percent])
factory.add_option("backpressure_inserter", [None, wave, intermittent_single_cycles,
    ↪ random_50_percent])
factory.generate_tests()

```

This will generate 32 tests (named `run_test_001` to `run_test_032`) with all possible permutations of the options provided for each argument. Note that we utilize some of the built-in generators to toggle backpressure and insert idle cycles.

## TUTORIAL: PING

One of the benefits of Python is the ease with which interfacing is possible. In this tutorial we'll look at interfacing the standard GNU `ping` command to the simulator. Using Python we can ping our DUT with fewer than 50 lines of code.

For the impatient this tutorial is provided as an example with `cocotb`. You can run this example from a fresh checkout:

```
cd examples/ping_tun_tap/tests
sudo make
```

---

**Note:** To create a virtual interface the test either needs root permissions or have `CAP_NET_ADMIN` capability.

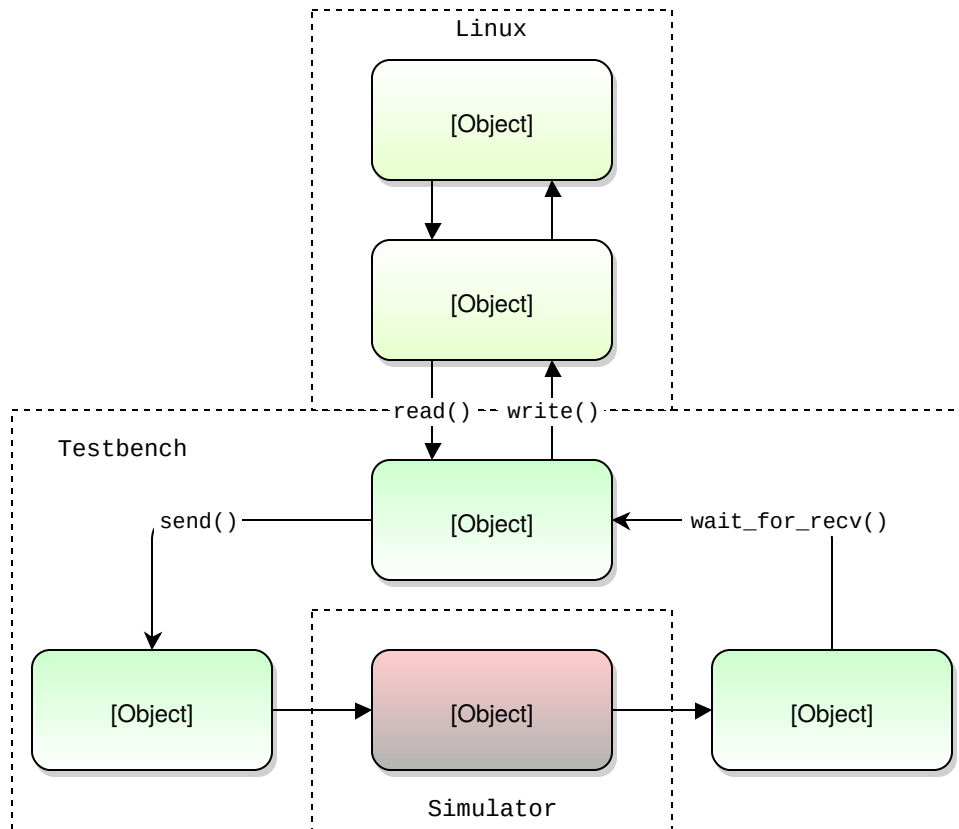
---

### 10.1 Architecture

We have a simple RTL block that takes ICMP echo requests and generates an ICMP echo response. To verify this behavior we want to run the `ping` utility against our RTL running in the simulator.

In order to achieve this we need to capture the packets that are created by ping, drive them onto the pins of our DUT in simulation, monitor the output of the DUT and send any responses back to the ping process.

Linux has a `TUN/TAP` virtual network device which we can use for this purpose, allowing `ping` to run unmodified and unaware that it is communicating with our simulation rather than a remote network endpoint.



## 10.2 Implementation

First of all we need to work out how to create a virtual interface. Python has a huge developer base and a quick search of the web reveals a [TUN example](#) that looks like an ideal starting point for our testbench. Using this example we write a function that will create our virtual interface:

```
import subprocess, fcntl, struct

def create_tun(name="tun0", ip="192.168.255.1"):
    TUNSETIFF = 0x400454ca
    TUNSETOWNER = TUNSETIFF + 2
    IFF_TUN = 0x0001
    IFF_NO_PI = 0x1000
    tun = open('/dev/net/tun', 'r+b')
    ifr = struct.pack('16sH', name, IFF_TUN | IFF_NO_PI)
    fcntl.ioctl(tun, TUNSETIFF, ifr)
    fcntl.ioctl(tun, TUNSETOWNER, 1000)
    subprocess.check_call('ifconfig tun0 %s up pointopoint 192.168.255.2 up' % ip,
        ↪ shell=True)
    return tun
```

Now we can get started on the actual test. First of all we'll create a clock signal and connect up the *Avalon driver* and *monitor* to the DUT. To help debug the testbench we'll enable verbose debug on the drivers and monitors by setting the log level to `logging.DEBUG`.



```

import cocotb
from cocotb.clock import Clock
from cocotb.drivers.avalon import AvalonSTPkts as AvalonSTDriver
from cocotb.monitors.avalon import AvalonSTPkts as AvalonSTMonitor

@cocotb.test()
def tun_tap_example_test(dut):
    cocotb.fork(Clock(dut.clk, 5000).start())

    stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)
    stream_out = AvalonSTMonitor(dut, "stream_out", dut.clk)

    # Enable verbose logging on the streaming interfaces
    stream_in.log.setLevel(logging.DEBUG)
    stream_out.log.setLevel(logging.DEBUG)

```

We also need to reset the DUT and drive some default values onto some of the bus signals. Note that we'll need to import the *Timer* and *RisingEdge* triggers.

```

# Reset the DUT
dut._log.debug("Resetting DUT")
dut.reset_n <= 0
stream_in.bus.valid <= 0
yield Timer(10, units='ns')
yield RisingEdge(dut.clk)
dut.reset_n <= 1
dut.stream_out_ready <= 1

```

The rest of the test becomes fairly straightforward. We create our TUN interface using our function defined previously. We'll also use the `subprocess` module to actually start the ping command.

We then wait for a packet by calling a blocking read call on the TUN file descriptor and simply append that to the queue on the driver. We wait for a packet to arrive on the monitor by yielding on `wait_for_recv()` and then write the received packet back to the TUN file descriptor.

```

# Create our interface (destroyed at the end of the test)
tun = create_tun()
fd = tun.fileno()

# Kick off a ping...
subprocess.check_call('ping -c 5 192.168.255.2 &', shell=True)

# Respond to 5 pings, then quit
for i in range(5):

    cocotb.log.info("Waiting for packets on tun interface")
    packet = os.read(fd, 2048)
    cocotb.log.info("Received a packet!")

    stream_in.append(packet)
    result = yield stream_out.wait_for_recv()

    os.write(fd, str(result))

```

That's it - simple!

## 10.3 Further work

This example is deliberately simplistic to focus on the fundamentals of interfacing to the simulator using TUN/TAP. As an exercise for the reader a useful addition would be to make the file descriptor non-blocking and spawn out separate coroutines for the monitor / driver, thus decoupling the sending and receiving of packets.

## TUTORIAL: DRIVER COSIMULATION

Cocotb was designed to provide a common platform for hardware and software developers to interact. By integrating systems early, ideally at the block level, it's possible to find bugs earlier in the design process.

For any given component that has a software interface there is typically a software abstraction layer or driver which communicates with the hardware. In this tutorial we will call unmodified production software from our testbench and re-use the code written to configure the entity.

For the impatient this tutorial is provided as an example with cocotb. You can run this example from a fresh checkout:

```
cd examples/endian_swapper/tests
make MODULE=test_endian_swapper_hal
```

---

**Note:** [SWIG](#) is required to compile the example

---

### 11.1 Difficulties with Driver Co-simulation

Co-simulating *un-modified* production software against a block-level testbench is not trivial – there are a couple of significant obstacles to overcome.

#### 11.1.1 Calling the HAL from a test

Typically the software component (often referred to as a Hardware Abstraction Layer or HAL) is written in C. We need to call this software from our test written in Python. There are multiple ways to call C code from Python, in this tutorial we'll use [SWIG](#) to generate Python bindings for our HAL.

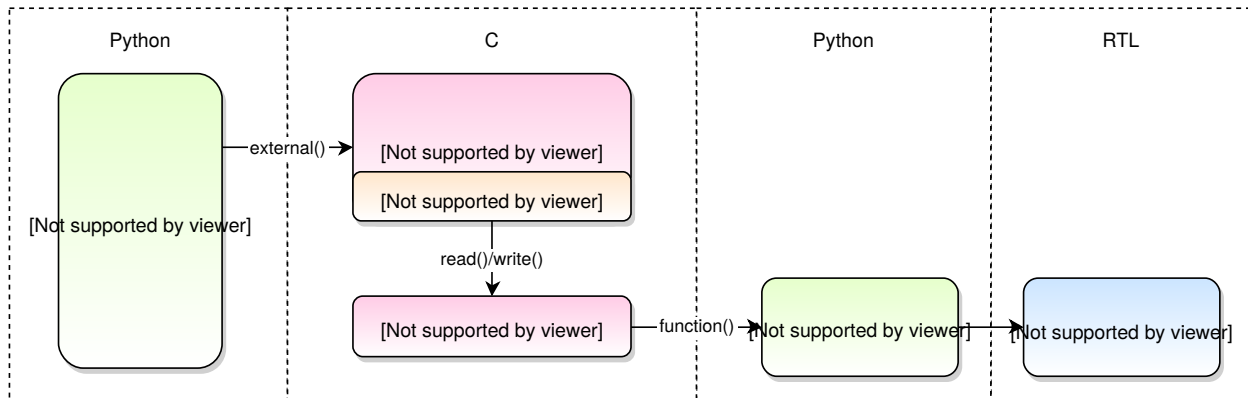
#### 11.1.2 Blocking in the driver

Another difficulty to overcome is the fact that the HAL is expecting to call a low-level function to access the hardware, often something like `ioread32`. We need this call to block while simulation time advances and a value is either read or written on the bus. To achieve this we link the HAL against a C library that provides the low level read/write functions. These functions in turn call into cocotb and perform the relevant access on the DUT.

### 11.2 Cocotb infrastructure

There are two decorators provided to enable this flow, which are typically used together to achieve the required functionality. The `cocotb.external` decorator turns a normal function that isn't a coroutine into a blocking

coroutine (by running the function in a separate thread). The `cocotb.function` decorator allows a *coroutine* that consumes simulation time to be called by a normal thread. The call sequence looks like this:



## 11.3 Implementation

### 11.3.1 Register Map

The endian swapper has a very simple register map:

Byte Offset	Register	Bits	Access	Description
0	CONTROL	0	R/W	Enable
		31:1	N/A	Reserved
4	PACKET_COUNT	31:0	RO	Number of Packets

### 11.3.2 HAL

To keep things simple we use the same RTL from the *Tutorial: Endian Swapper*. We write a simplistic HAL which provides the following functions:

```
endian_swapper_enable(endian_swapper_state_t *state);
endian_swapper_disable(endian_swapper_state_t *state);
endian_swapper_get_count(endian_swapper_state_t *state);
```

These functions call `IORD` and `IOWR` – usually provided by the Altera NIOS framework.

### 11.3.3 IO Module

This module acts as the bridge between the C HAL and the Python testbench. It exposes the `IORD` and `IOWR` calls to link the HAL against, but also provides a Python interface to allow the read/write bindings to be dynamically set (through `set_write_function` and `set_read_function` module functions).

In a more complicated scenario, this could act as an interconnect, dispatching the access to the appropriate driver depending on address decoding, for instance.

### 11.3.4 Testbench

First of all we set up a clock, create an *Avalon Master* interface and reset the DUT. Then we create two functions that are wrapped with the *cocotb.function* decorator to be called when the HAL attempts to perform a read or write. These are then passed to the *IO Module*:

```
@cocotb.function
def read(address):
    master.log.debug("External source: reading address 0x%08X" % address)
    value = yield master.read(address)
    master.log.debug("Reading complete: got value 0x%08x" % value)
    raise ReturnValue(value)

@cocotb.function
def write(address, value):
    master.log.debug("Write called for 0x%08X -> %d" % (address, value))
    yield master.write(address, value)
    master.log.debug("Write complete")

io_module.set_write_function(write)
io_module.set_read_function(read)
```

We can then initialize the HAL and call functions, using the *cocotb.external* decorator to turn the normal function into a blocking coroutine that we can yield:

```
state = hal.endian_swapper_init(0)
yield cocotb.external(hal.endian_swapper_enable)(state)
```

The HAL will perform whatever calls it needs, accessing the DUT through the *Avalon-MM driver*, and control will return to the testbench when the function returns.

---

**Note:** The decorator is applied to the function before it is called.

---

## 11.4 Further Work

In future tutorials we'll consider co-simulating unmodified drivers written using mmap (for example built upon the *UIO framework*) and consider interfacing with emulators like *QEMU* to allow us to co-simulate when the software needs to execute on a different processor architecture.



## MORE EXAMPLES

Apart from the examples covered with full tutorials in the previous sections, the directory `cocotb/examples/` contains some more smaller modules you may want to take a look at.

### 12.1 Adder

The directory `cocotb/examples/adder/` contains an adder RTL in both Verilog and VHDL, an `adder_model` implemented in Python, and the cocotb testbench with two defined tests: a simple `adder_basic_test()` and a slightly more advanced `adder_randomised_test()`.

This example does not use any *Driver*, *Monitor*, or *Scoreboard*; not even a clock.

### 12.2 D Flip-Flop

The directory `cocotb/examples/dff/` contains a simple D flip-flop, implemented in both VHDL and Verilog.

The HDL has the data input port `d`, the clock port `c`, and the data output `q` with an initial state of 0. No reset port exists.

The cocotb testbench checks the initial state first, then applies random data to the data input. The flip-flop output is captured at each rising edge of the clock and compared to the applied input data using a *Scoreboard*.

The testbench defines a `BitMonitor` (a sub-class of *Monitor*) as a pendant to the cocotb-provided *BitDriver*. The *BitDriver*'s `start()` and `stop()` methods are used to start and stop generation of input data.

A *TestFactory* is used to generate the random tests.

### 12.3 Mean

The directory `cocotb/examples/mean/` contains a module that calculates the mean value of a data input bus `i` (with signals `i_data` and `i_valid`) and outputs it on `o` (with `i_data` and `o_valid`).

It has implementations in both VHDL and SystemVerilog.

The testbench defines a `StreamBusMonitor` (a sub-class of *BusMonitor*), a clock generator, a `value_test` helper coroutine and a few tests. Test `mean_randomised_test` uses the `StreamBusMonitor` to feed a *Scoreboard* with the collected transactions on input bus `i`.

## 12.4 Mixed Language

The directory `cocotb/examples/mixed_language/` contains two toplevel HDL files, one in VHDL, one in SystemVerilog, that each instantiate the `endian_swapper` in SystemVerilog and VHDL in parallel and chains them together so that the endianness is swapped twice.

Thus, we end up with SystemVerilog+VHDL instantiated in VHDL and SystemVerilog+VHDL instantiated in SystemVerilog.

The cocotb testbench pulls the reset on both instances and checks that they behave the same.

---

**Todo:** This example is not complete.

---

## 12.5 AXI Lite Slave

The directory `cocotb/examples/axi_lite_slave/` contains ...

---

**Todo:** Write documentation, see `README.md`

---

## 12.6 Sorter

Example testbench for snippet of code from `comp.lang.verilog`:

```
@cocotb.coroutine
def run_test(dut, data_generator=random_data, delay_cycles=2):
    """Send data through the DUT and check it is sorted output."""
    cocotb.fork(Clock(dut.clk, 100).start())

    # Don't check until valid output
    expected = [None] * delay_cycles

    for index, values in enumerate(data_generator(bits=len(dut.in1))):
        expected.append(sorted(values))

    yield RisingEdge(dut.clk)
    dut.in1 = values[0]
    dut.in2 = values[1]
    dut.in3 = values[2]
    dut.in4 = values[3]
    dut.in5 = values[4]

    yield ReadOnly()
    expect = expected.pop(0)

    if expect is None:
        continue

    got = [int(dut.out5), int(dut.out4), int(dut.out3),
           int(dut.out2), int(dut.out1)]
```

(continues on next page)



(continued from previous page)

```
    if got != expect:
        dut._log.error('Expected %s' % expect)
        dut._log.error('Got %s' % got)
        raise TestFailure("Output didn't match")

dut._log.info('Sucessfully sent %d cycles of data' % (index + 1))
```



## TROUBLESHOOTING

### 13.1 Simulation Hangs

Did you call a function that is marked as a coroutine directly, i.e. without using `yield`?

### 13.2 Increasing Verbosity

If things fail in the VPI/VHPI/FLI area, check your simulator's documentation to see if it has options to increase its verbosity about what may be wrong. You can then set these options on the `make` command line as `COMPILE_ARGS`, `SIM_ARGS` or `EXTRA_OPTS` (see *Build options and Environment Variables* for details).

### 13.3 Attaching a Debugger

In order to give yourself time to attach a debugger to the simulator process before it starts to run, you can set the environment variable `COCOTB_ATTACH` to a pause time value in seconds. If set, `cocotb` will print the process ID (PID) to attach to and wait the specified time before actually letting the simulator run.

For the GNU debugger GDB, the command is `attach <process-id>`.



## SIMULATOR SUPPORT

This page documents any known quirks and gotchas in the various simulators.

### 14.1 Icarus

#### 14.1.1 Accessing bits in a vector

Accessing bits of a vector doesn't work:

```
dut.stream_in_data[2] <= 1
```

See `access_single_bit` test in `examples/functionality/tests/test_discovery.py`.

#### 14.1.2 Wavefoms

To get waveform in VCD format some Verilog code must be added in the top component as example below:

```
module button_deb(
    input  clk,
    input  rst,
    input  button_in,
    output button_valid);

//... verilog module code here

// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
    $dumpfile ("button_deb.vcd");
    $dumpvars (0, button_deb);
    #1;
end
`endif
endmodule
```

## 14.2 Synopsys VCS

## 14.3 Aldec Riviera-PRO

The `$LICENSE_QUEUE` environment variable can be used for this simulator – this setting will be mirrored in the TCL `license_queue` variable to control runtime license checkouts.

## 14.4 Mentor Questa

## 14.5 Mentor ModelSim

Any ModelSim PE or ModelSim PE derivative (like ModelSim Microsemi, Intel, Lattice Edition) does not support the VHDL FLI feature. If you try to run with FLI enabled, you will see a `vsim-FLI-3155` error:

```
** Error (suppressible): (vsim-FLI-3155) The FLI is not enabled in this version of ↵  
↵ModelSim.
```

ModelSim DE and SE (and Questa, of course) supports the FLI.

## 14.6 Cadence Incisive, Cadence Xcelium

## 14.7 GHDL

Support is preliminary. Noteworthy is that despite GHDL being a VHDL simulator, it implements the VPI interface.

## ROADMAP

cocotb is in active development.

We use GitHub issues to track our pending tasks. Take a look at the [open Feature List](#) to see the work that's lined up.

If you have a GitHub account you can also [raise an enhancement request](#) to suggest new features.





## RELEASE NOTES

All releases are available from the [GitHub Releases Page](#).

### 16.1 cocotb 1.2

Released on 24 July 2019

#### 16.1.1 New features

- cocotb is now built as Python package and installable through pip. ([#517](#), [#799](#), [#800](#), [#803](#), [#805](#))
- Support for `async` functions and generators was added (Python 3 only). Please have a look at *Async functions* for an example how to use this new feature.
- VHDL block statements can be traversed. ([#850](#))
- Support for Python 3.7 was added.

#### 16.1.2 Notable changes and bug fixes

- The heart of cocotb, its scheduler, is now even more robust. Many small bugs, inconsistencies and unreliable behavior have been ironed out.
- Exceptions are now correctly propagated between coroutines, giving users the “natural” behavior they’d expect with exceptions. ([#633](#))
- The `setimmediatevalue()` function now works for values larger than 32 bit. ([#768](#))
- The documentation was cleaned up, improved and extended in various places, making it more consistent and complete.
- Tab completion in newer versions of IPython is fixed. ([#825](#))
- Python 2.6 is officially not supported any more. cocotb supports Python 2.7 and Python 3.5+.
- The cocotb GitHub project moved from `potentialventures/cocotb` to `cocotb/cocotb`. Redirects for old URLs are in place.

### 16.1.3 Known issues

- Depending on your simulation, cocotb 1.2 might be roughly 20 percent slower than cocotb 1.1. Much of the work in this release cycle went into fixing correctness bugs in the scheduler, sometimes at the cost of performance. We are continuing to investigate this in [issue #961](#). Independent of the cocotb version, we recommend using the latest Python 3 version, which is shown to be significantly faster than previous Python 3 versions, and slightly faster than Python 2.7.

Please have a look at the [issue tracker](#) for more outstanding issues and contribution opportunities.

## 16.2 cocotb 1.1

Released on 24 January 2019.

This release is the result of four years of work with too many bug fixes, improvements and refactorings to name them all. Please have a look at the release announcement [on the mailing list](#) for further information.

## 16.3 cocotb 1.0

Released on 15 February 2015.

### 16.3.1 New features

- FLI support for ModelSim
- Mixed Language, Verilog and VHDL
- Windows
- 300% performance improvement with VHPI interface
- WaveDrom support for wave diagrams.

## 16.4 cocotb 0.4

Released on 25 February 2014.

### 16.4.1 New features

- Issue [#101](#): Implement Lock primitive to support mutex
- Issue [#105](#): Compatibility with Aldec Riviera-Pro
- Issue [#109](#): Combine multiple `results.xml` into a single results file
- Issue [#111](#): XGMII drivers and monitors added
- Issue [#113](#): Add operators to `BinaryValue` class
- Issue [#116](#): Native VHDL support by implementing VHPI layer
- Issue [#117](#): Added AXI4-Lite Master BFM

## 16.4.2 Bugs fixed

- Issue [#100](#): Functional bug in endian\_swapper example RTL
- Issue [#102](#): Only 1 coroutine wakes up of multiple coroutines wait() on an Event
- Issue [#114](#): Fix build issues with Cadence IUS simulator

## 16.4.3 New examples

- Issue [#106](#): TUN/TAP example using ping

## 16.5 cocotb 0.3

Released on 27 September 2013.

This contains a raft of fixes and feature enhancements.

## 16.6 cocotb 0.2

Released on 19 July 2013.

### 16.6.1 New features

- Release 0.2 supports more simulators and increases robustness over 0.1.
- A centralized installation is now supported (see documentation) with supporting libraries build when the simulation is run for the first time.

## 16.7 cocotb 0.1

Released on 9 July 2013.

- The first release of cocotb.
- Allows installation and running against Icarus, VCS, Aldec simulators.



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### C

`cocotb.handle`, [41](#)  
`cocotb.result`, [27](#)  
`cocotb.scoreboard`, [36](#)  
`cocotb.utils`, [38](#)  
`cocotb.wavedrom`, [48](#)





## Symbols

\_\_check\_vhpi\_error (C++ function), 70  
 \_\_check\_vpi\_error (C++ function), 76  
 \_\_gpi\_get\_handle\_by\_name (C++ function), 68  
 \_\_gpi\_get\_handle\_by\_raw (C++ function), 68  
 \_driver\_send (cocotb.drivers.BusDriver attribute), 35  
 \_driver\_send() (cocotb.drivers.Driver method), 34  
 \_log\_level\_table (C++ class), 82, 92  
 \_log\_level\_table::level (C++ member), 82  
 \_log\_level\_table::levelname (C++ member), 82  
 \_monitor\_recv (cocotb.monitors.Monitor attribute), 36  
 \_next\_valids() (cocotb.drivers.ValidatedBusDriver method), 35  
 \_recv() (cocotb.monitors.Monitor method), 36  
 \_send (cocotb.drivers.Driver attribute), 34  
 \_state (C++ member), 92  
 \_wait\_for\_nsignal (cocotb.drivers.BusDriver attribute), 35  
 \_wait\_for\_signal (cocotb.drivers.BusDriver attribute), 35

## A

acquire() (cocotb.triggers.Lock method), 22  
 AD9361 (class in cocotb.drivers.ad9361), 43  
 ad9361\_tx\_to\_rx\_loopback() (cocotb.drivers.ad9361.AD9361 method), 44  
 add\_interface() (cocotb.scoreboard.Scoreboard method), 37  
 add\_module\_constants (C++ function), 89  
 add\_option() (cocotb.regression.TestFactory method), 29  
 append() (cocotb.drivers.Driver method), 33  
 argv (C++ member), 81  
 assign() (cocotb.binary.BinaryValue method), 30  
 AssignmentResult (class in cocotb.handle), 41  
 AvalonMaster (class in cocotb.drivers.avalon), 45  
 AvalonMemory (class in cocotb.drivers.avalon), 46  
 AvalonMM (class in cocotb.drivers.avalon), 45  
 AvalonST (class in cocotb.drivers.avalon), 46

AvalonST (class in cocotb.monitors.avalon), 47  
 AvalonSTPkts (class in cocotb.drivers.avalon), 46  
 AvalonSTPkts (class in cocotb.monitors.avalon), 47  
 AXI4LiteMaster (class in cocotb.drivers.amba), 44  
 AXI4Slave (class in cocotb.drivers.amba), 45

## B

BinaryRepresentation (class in cocotb.binary), 30  
 BinaryValue (class in cocotb.binary), 30  
 binstr() (cocotb.binary.BinaryValue property), 31  
 BitDriver (class in cocotb.drivers), 34  
 buff() (cocotb.binary.BinaryValue property), 31  
 Bus (class in cocotb.bus), 31  
 BusDriver (class in cocotb.drivers), 34  
 BusMonitor (class in cocotb.monitors), 36

## C

cache\_time (C++ member), 90  
 capture() (cocotb.bus.Bus method), 32  
 CASE\_OPTION (C macro), 67  
 CASE\_STR (C macro), 69, 75  
 CHECK\_AND\_STORE (C macro), 67  
 check\_vhpi\_error (C macro), 70  
 check\_vpi\_error (C macro), 75  
 clear() (cocotb.drivers.Driver method), 33  
 clear() (cocotb.triggers.Event method), 21  
 clear() (cocotb.wavedrom.Wavedrom method), 48  
 Clock (class in cocotb.clock), 32, 38  
 ClockCycles (class in cocotb.triggers), 19  
 cocotb.handle (module), 41  
 cocotb.result (module), 27  
 cocotb.scoreboard (module), 36  
 cocotb.utils (module), 38  
 cocotb.wavedrom (module), 48  
 COCOTB\_ACTIVE\_ID (C macro), 90  
 COCOTB\_ATTACH, 13  
 cocotb\_entrypoint (C++ class), 56, 80  
 cocotb\_entrypoint::cocotb\_arch (C++ class), 56, 80  
 COCOTB\_INACTIVE\_ID (C macro), 90  
 cocotb\_init (C++ function), 57  
 COCOTB\_MODULE (C macro), 81

COCOTB\_NVC\_TRACE, [12](#)  
COCOTB\_RESULTS\_FILE, [11](#)  
Combine (*class in cocotb.triggers*), [20](#)  
compare() (*cocotb.scoreboard.Scoreboard method*),  
[37](#)  
COMPILE\_ARGS, [11](#)  
ConstantObject (*class in cocotb.handle*), [41](#)  
coroutine (*class in cocotb*), [28](#)  
create\_error() (*in module cocotb.result*), [27](#)  
CUSTOM\_COMPILE\_DEPS, [12](#)  
CUSTOM\_SIM\_DEPS, [12](#)

## D

deregister\_callback (C++ *function*), [89](#), [91](#)  
DOT\_LIB\_EXT (C *macro*), [67](#)  
drive() (*cocotb.bus.Bus method*), [32](#)  
Driver (*class in cocotb.drivers*), [33](#)  
drivers() (*cocotb.handle.NonConstantObject method*), [42](#)  
DROP\_GIL (C++ *function*), [90](#)

## E

Edge (*class in cocotb.triggers*), [19](#)  
embed\_init\_python (C++ *function*), [81](#)  
embed\_sim\_event (C++ *function*), [81](#)  
embed\_sim\_init (C++ *function*), [81](#)  
END (C++ *enumerator*), [86](#)  
EnumObject (*class in cocotb.handle*), [42](#)  
environment variable  
    COCOTB\_ANSI\_OUTPUT, [12](#)  
    COCOTB\_ATTACH, [13](#)  
    COCOTB\_ENABLE\_PROFILING, [13](#)  
    COCOTB\_HOOKS, [13](#)  
    COCOTB\_LOG\_LEVEL, [13](#)  
    COCOTB\_PY\_DIR, [13](#)  
    COCOTB\_REDUCED\_LOG\_FMT, [12](#)  
    COCOTB\_RESOLVE\_X, [13](#)  
    COCOTB\_RESULTS\_FILE, [11](#), [12](#)  
    COCOTB\_SCHEDULER\_DEBUG, [13](#)  
    COCOTB\_SHARE\_DIR, [13](#)  
    COVERAGE, [13](#)  
    MEMCHECK, [13](#)  
    MODULE, [12](#)  
    RANDOM\_SEED, [12](#)  
    TESTCASE, [12](#)  
    TOPELVEL, [12](#)  
    VERSION, [13](#)  
error\_out (C++ *function*), [90](#), [92](#)  
Event (*class in cocotb.triggers*), [21](#)  
external (*class in cocotb*), [28](#)  
ExternalException, [27](#)  
EXTRA\_ARGS, [11](#)

## F

FallingEdge (*class in cocotb.triggers*), [19](#)  
First (*class in cocotb.triggers*), [20](#)  
fli\_mappings (C++ *function*), [57](#)  
fli\_table (C++ *member*), [57](#)  
FliEnumObjHdl (C++ *class*), [49](#), [62](#)  
FliEnumObjHdl::~~FliEnumObjHdl (C++ *function*), [62](#)  
FliEnumObjHdl::FliEnumObjHdl (C++ *function*), [62](#)  
FliEnumObjHdl::get\_signal\_value\_long (C++ *function*), [62](#)  
FliEnumObjHdl::get\_signal\_value\_str (C++ *function*), [62](#)  
FliEnumObjHdl::initialise (C++ *function*), [62](#)  
FliEnumObjHdl::m\_num\_enum (C++ *member*), [62](#)  
FliEnumObjHdl::m\_value\_enum (C++ *member*), [62](#)  
FliEnumObjHdl::set\_signal\_value (C++ *function*), [62](#)  
FliImpl (C++ *class*), [49](#), [65](#)  
FliImpl::cache (C++ *member*), [67](#)  
FliImpl::create\_gpi\_obj\_from\_handle (C++ *function*), [66](#)  
FliImpl::deregister\_callback (C++ *function*), [66](#)  
FliImpl::FliImpl (C++ *function*), [66](#)  
FliImpl::get\_root\_handle (C++ *function*), [66](#)  
FliImpl::get\_sim\_precision (C++ *function*), [66](#)  
FliImpl::get\_sim\_time (C++ *function*), [66](#)  
FliImpl::isTypeSignal (C++ *function*), [67](#)  
FliImpl::isTypeValue (C++ *function*), [67](#)  
FliImpl::isValueBoolean (C++ *function*), [67](#)  
FliImpl::isValueChar (C++ *function*), [67](#)  
FliImpl::isValueConst (C++ *function*), [67](#)  
FliImpl::isValueLogic (C++ *function*), [67](#)  
FliImpl::iterate\_handle (C++ *function*), [66](#)  
FliImpl::m\_nexttime\_cbhdl (C++ *member*), [67](#)  
FliImpl::m\_readonly\_cbhdl (C++ *member*), [67](#)  
FliImpl::m\_readwrite\_cbhdl (C++ *member*), [67](#)  
FliImpl::native\_check\_create (C++ *function*), [66](#)  
FliImpl::reason\_to\_string (C++ *function*), [66](#)  
FliImpl::register\_nexttime\_callback (C++ *function*), [66](#)  
FliImpl::register\_readonly\_callback (C++ *function*), [66](#)  
FliImpl::register\_readwrite\_callback (C++ *function*), [66](#)  
FliImpl::register\_timed\_callback (C++ *function*), [66](#)  
FliImpl::sim\_end (C++ *function*), [66](#)

FliIntObjHdl (C++ class), 50, 63  
 FliIntObjHdl::~~FliIntObjHdl (C++ function), 63  
 FliIntObjHdl::FliIntObjHdl (C++ function), 63  
 FliIntObjHdl::get\_signal\_value\_binstr (C++ function), 63  
 FliIntObjHdl::get\_signal\_value\_long (C++ function), 63  
 FliIntObjHdl::initialise (C++ function), 63  
 FliIntObjHdl::set\_signal\_value (C++ function), 63  
 FliIterator (C++ class), 50, 64  
 FliIterator::~~FliIterator (C++ function), 65  
 FliIterator::FliIterator (C++ function), 65  
 FliIterator::iterate\_over (C++ member), 65  
 FliIterator::m\_currentHandles (C++ member), 65  
 FliIterator::m\_iterator (C++ member), 65  
 FliIterator::m\_regs (C++ member), 65  
 FliIterator::m\_sigs (C++ member), 65  
 FliIterator::m\_vars (C++ member), 65  
 FliIterator::next\_handle (C++ function), 65  
 FliIterator::one2many (C++ member), 65  
 FliIterator::populate\_handle\_list (C++ function), 65  
 FliIterator::selected (C++ member), 65  
 FliLogicObjHdl (C++ class), 50, 62  
 FliLogicObjHdl::~~FliLogicObjHdl (C++ function), 63  
 FliLogicObjHdl::FliLogicObjHdl (C++ function), 63  
 FliLogicObjHdl::get\_signal\_value\_binstr (C++ function), 63  
 FliLogicObjHdl::initialise (C++ function), 63  
 FliLogicObjHdl::m\_enum\_map (C++ member), 63  
 FliLogicObjHdl::m\_mti\_buff (C++ member), 63  
 FliLogicObjHdl::m\_num\_enum (C++ member), 63  
 FliLogicObjHdl::m\_value\_enum (C++ member), 63  
 FliLogicObjHdl::set\_signal\_value (C++ function), 63  
 FliNextPhaseCbHdl (C++ class), 50, 59  
 FliNextPhaseCbHdl::~~FliNextPhaseCbHdl (C++ function), 59  
 FliNextPhaseCbHdl::FliNextPhaseCbHdl (C++ function), 59  
 FliObj (C++ class), 50, 60  
 FliObj::~~FliObj (C++ function), 60  
 FliObj::FliObj (C++ function), 60  
 FliObj::get\_acc\_full\_type (C++ function), 60  
 FliObj::get\_acc\_type (C++ function), 60  
 FliObj::m\_acc\_full\_type (C++ member), 61  
 FliObj::m\_acc\_type (C++ member), 61  
 FliObjHdl (C++ class), 50, 61  
 FliObjHdl::~~FliObjHdl (C++ function), 61  
 FliObjHdl::FliObjHdl (C++ function), 61  
 FliObjHdl::initialise (C++ function), 61  
 FliProcessCbHdl (C++ class), 51, 57  
 FliProcessCbHdl::~~FliProcessCbHdl (C++ function), 58  
 FliProcessCbHdl::arm\_callback (C++ function), 58  
 FliProcessCbHdl::cleanup\_callback (C++ function), 57  
 FliProcessCbHdl::FliProcessCbHdl (C++ function), 58  
 FliProcessCbHdl::m\_proc\_hdl (C++ member), 58  
 FliProcessCbHdl::m\_sensitised (C++ member), 58  
 FliReadOnlyCbHdl (C++ class), 51, 59  
 FliReadOnlyCbHdl::~~FliReadOnlyCbHdl (C++ function), 59  
 FliReadOnlyCbHdl::FliReadOnlyCbHdl (C++ function), 59  
 FliReadWriteCbHdl (C++ class), 51, 59  
 FliReadWriteCbHdl::~~FliReadWriteCbHdl (C++ function), 59  
 FliReadWriteCbHdl::FliReadWriteCbHdl (C++ function), 59  
 FliRealObjHdl (C++ class), 51, 63  
 FliRealObjHdl::~~FliRealObjHdl (C++ function), 63  
 FliRealObjHdl::FliRealObjHdl (C++ function), 63  
 FliRealObjHdl::get\_signal\_value\_real (C++ function), 63  
 FliRealObjHdl::initialise (C++ function), 63  
 FliRealObjHdl::m\_mti\_buff (C++ member), 64  
 FliRealObjHdl::set\_signal\_value (C++ function), 63  
 FliShutdownCbHdl (C++ class), 51, 59  
 FliShutdownCbHdl::~~FliShutdownCbHdl (C++ function), 60  
 FliShutdownCbHdl::arm\_callback (C++ function), 60  
 FliShutdownCbHdl::FliShutdownCbHdl (C++ function), 60  
 FliShutdownCbHdl::run\_callback (C++ function), 60  
 FliSignalCbHdl (C++ class), 51, 58  
 FliSignalCbHdl::~~FliSignalCbHdl (C++ function), 58

`FliSignalCbHdl::arm_callback` (C++ *function*), 58

`FliSignalCbHdl::cleanup_callback` (C++ *function*), 58

`FliSignalCbHdl::FliSignalCbHdl` (C++ *function*), 58

`FliSignalCbHdl::m_sig_hdl` (C++ *member*), 58

`FliSignalObjHdl` (C++ *class*), 52, 61

`FliSignalObjHdl::~~FliSignalObjHdl` (C++ *function*), 61

`FliSignalObjHdl::FliSignalObjHdl` (C++ *function*), 61

`FliSignalObjHdl::initialise` (C++ *function*), 61

`FliSignalObjHdl::is_var` (C++ *function*), 61

`FliSignalObjHdl::m_either_cb` (C++ *member*), 61

`FliSignalObjHdl::m_falling_cb` (C++ *member*), 61

`FliSignalObjHdl::m_is_var` (C++ *member*), 61

`FliSignalObjHdl::m_rising_cb` (C++ *member*), 61

`FliSignalObjHdl::value_change_cb` (C++ *function*), 61

`FliSimPhaseCbHdl` (C++ *class*), 52, 58

`FliSimPhaseCbHdl::~~FliSimPhaseCbHdl` (C++ *function*), 59

`FliSimPhaseCbHdl::arm_callback` (C++ *function*), 58

`FliSimPhaseCbHdl::FliSimPhaseCbHdl` (C++ *function*), 59

`FliSimPhaseCbHdl::m_priority` (C++ *member*), 59

`FliStartupCbHdl` (C++ *class*), 52, 59

`FliStartupCbHdl::~~FliStartupCbHdl` (C++ *function*), 59

`FliStartupCbHdl::arm_callback` (C++ *function*), 59

`FliStartupCbHdl::FliStartupCbHdl` (C++ *function*), 59

`FliStartupCbHdl::run_callback` (C++ *function*), 59

`FliStringObjHdl` (C++ *class*), 52, 64

`FliStringObjHdl::~~FliStringObjHdl` (C++ *function*), 64

`FliStringObjHdl::FliStringObjHdl` (C++ *function*), 64

`FliStringObjHdl::get_signal_value_str` (C++ *function*), 64

`FliStringObjHdl::initialise` (C++ *function*), 64

`FliStringObjHdl::m_mti_buff` (C++ *member*), 64

`FliStringObjHdl::set_signal_value` (C++ *function*), 64

`FliTimedCbHdl` (C++ *class*), 52, 60

`FliTimedCbHdl::~~FliTimedCbHdl` (C++ *function*), 60

`FliTimedCbHdl::arm_callback` (C++ *function*), 60

`FliTimedCbHdl::cleanup_callback` (C++ *function*), 60

`FliTimedCbHdl::FliTimedCbHdl` (C++ *function*), 60

`FliTimedCbHdl::m_time_ps` (C++ *member*), 60

`FliTimedCbHdl::reset_time` (C++ *function*), 60

`FliTimerCache` (C++ *class*), 52, 64

`FliTimerCache::~~FliTimerCache` (C++ *function*), 64

`FliTimerCache::FliTimerCache` (C++ *function*), 64

`FliTimerCache::free_list` (C++ *member*), 64

`FliTimerCache::get_timer` (C++ *function*), 64

`FliTimerCache::impl` (C++ *member*), 64

`FliTimerCache::put_timer` (C++ *function*), 64

`FliValueObjHdl` (C++ *class*), 53, 61

`FliValueObjHdl::~~FliValueObjHdl` (C++ *function*), 61

`FliValueObjHdl::FliValueObjHdl` (C++ *function*), 61

`FliValueObjHdl::get_fli_typeid` (C++ *function*), 62

`FliValueObjHdl::get_fli_typekind` (C++ *function*), 62

`FliValueObjHdl::get_signal_value_binstr` (C++ *function*), 61

`FliValueObjHdl::get_signal_value_long` (C++ *function*), 62

`FliValueObjHdl::get_signal_value_real` (C++ *function*), 62

`FliValueObjHdl::get_signal_value_str` (C++ *function*), 61

`FliValueObjHdl::get_sub_hdl` (C++ *function*), 62

`FliValueObjHdl::initialise` (C++ *function*), 62

`FliValueObjHdl::m_fli_type` (C++ *member*), 62

`FliValueObjHdl::m_sub_hdls` (C++ *member*), 62

`FliValueObjHdl::m_val_buff` (C++ *member*), 62

`FliValueObjHdl::m_val_type` (C++ *member*), 62

`FliValueObjHdl::set_signal_value` (C++ *function*), 62

*function (class in cocotb)*, 28

## G

- GEN\_IDX\_SEP\_LHS (*C macro*), 70
- GEN\_IDX\_SEP\_RHS (*C macro*), 70
- generate\_tests() (*cocotb.regression.TestFactory method*), 29
- get() (*cocotb.wavedrom.Wavedrom method*), 48
- get\_binstr() (*cocotb.binary.BinaryValue method*), 31
- get\_buff() (*cocotb.binary.BinaryValue method*), 31
- get\_const(*C++ function*), 89, 91
- get\_definition\_file(*C++ function*), 89, 91
- get\_definition\_name(*C++ function*), 89, 91
- get\_handle\_by\_index(*C++ function*), 89, 91
- get\_handle\_by\_name(*C++ function*), 89, 91
- get\_module\_ref(*C++ function*), 81
- get\_name\_string(*C++ function*), 89, 91
- get\_num\_elems(*C++ function*), 89, 91
- get\_precision(*C++ function*), 89, 91
- get\_range(*C++ function*), 69, 89, 91
- get\_root\_handle(*C++ function*), 89, 91
- get\_signal\_val\_binstr(*C++ function*), 89, 91
- get\_signal\_val\_long(*C++ function*), 89, 90
- get\_signal\_val\_real(*C++ function*), 89, 90
- get\_signal\_val\_str(*C++ function*), 89, 90
- get\_sim\_steps() (*in module cocotb.utils*), 39
- get\_sim\_time(*C++ function*), 89, 91
- get\_sim\_time() (*in module cocotb.utils*), 38
- get\_time\_from\_sim\_steps() (*in module cocotb.utils*), 38
- get\_type(*C++ function*), 89, 91
- get\_type\_string(*C++ function*), 89, 91
- get\_value() (*cocotb.binary.BinaryValue method*), 31
- get\_value\_signed() (*cocotb.binary.BinaryValue method*), 31
- GETSTATE (*C macro*), 88
- GPI\_CALL(*C++ enumerator*), 83
- gpi\_cb\_state(*C++ enum*), 83
- gpi\_cb\_state\_e(*C++ type*), 82
- gpi\_create\_clock(*C++ function*), 69
- GPI\_DELETE(*C++ enumerator*), 83
- gpi\_deregister\_callback(*C++ function*), 69
- gpi\_embed\_end(*C++ function*), 67, 83
- gpi\_embed\_event(*C++ function*), 67, 83
- gpi\_embed\_init(*C++ function*), 67, 83
- GPI\_ENTRY\_POINT (*C macro*), 82
- GPI\_ENTRY\_POINT(*C++ function*), 57
- GPI\_FREE(*C++ enumerator*), 83
- gpi\_function\_t(*C++ type*), 90
- gpi\_get\_definition\_file(*C++ function*), 68
- gpi\_get\_definition\_name(*C++ function*), 68
- gpi\_get\_handle\_by\_index(*C++ function*), 68
- gpi\_get\_handle\_by\_name(*C++ function*), 68
- gpi\_get\_num\_elems(*C++ function*), 68
- gpi\_get\_object\_type(*C++ function*), 68
- gpi\_get\_range\_left(*C++ function*), 68
- gpi\_get\_range\_right(*C++ function*), 68
- gpi\_get\_root\_handle(*C++ function*), 68
- gpi\_get\_signal\_name\_str(*C++ function*), 68
- gpi\_get\_signal\_type\_str(*C++ function*), 68
- gpi\_get\_signal\_value\_binstr(*C++ function*), 68
- gpi\_get\_signal\_value\_long(*C++ function*), 68
- gpi\_get\_signal\_value\_real(*C++ function*), 68
- gpi\_get\_signal\_value\_str(*C++ function*), 68
- gpi\_get\_sim\_precision(*C++ function*), 68
- gpi\_get\_sim\_time(*C++ function*), 68
- gpi\_is\_constant(*C++ function*), 68
- gpi\_is\_indexable(*C++ function*), 68
- gpi\_iterate(*C++ function*), 68
- gpi\_iterator\_hdl\_converter(*C++ function*), 90
- gpi\_load\_extra\_libs(*C++ function*), 68, 83
- gpi\_load\_libs(*C++ function*), 67
- gpi\_log(*C++ function*), 82
- gpi\_next(*C++ function*), 68
- GPI\_PRIMED(*C++ enumerator*), 83
- gpi\_print\_registered\_impl(*C++ function*), 67
- gpi\_register\_impl(*C++ function*), 67, 83
- gpi\_register\_nexttime\_callback(*C++ function*), 68
- gpi\_register\_readonly\_callback(*C++ function*), 68
- gpi\_register\_readwrite\_callback(*C++ function*), 68
- gpi\_register\_timed\_callback(*C++ function*), 68
- gpi\_register\_value\_change\_callback(*C++ function*), 68
- GPI\_REPRIME(*C++ enumerator*), 83
- gpi\_set\_signal\_value\_long(*C++ function*), 68
- gpi\_set\_signal\_value\_real(*C++ function*), 68
- gpi\_set\_signal\_value\_str(*C++ function*), 68
- gpi\_sim\_end(*C++ function*), 67
- gpi\_sim\_hdl\_converter(*C++ function*), 90
- gpi\_stop\_clock(*C++ function*), 69
- GpiCbHdl(*C++ class*), 53, 85
- GpiCbHdl::~~GpiCbHdl(*C++ function*), 85
- GpiCbHdl::arm\_callback(*C++ function*), 85
- GpiCbHdl::cleanup\_callback(*C++ function*), 85
- GpiCbHdl::get\_call\_state(*C++ function*), 85
- GpiCbHdl::get\_user\_data(*C++ function*), 85
- GpiCbHdl::gpi\_function(*C++ member*), 86
- GpiCbHdl::GpiCbHdl(*C++ function*), 85
- GpiCbHdl::m\_cb\_data(*C++ member*), 86
- GpiCbHdl::m\_state(*C++ member*), 86
- GpiCbHdl::run\_callback(*C++ function*), 85
- GpiCbHdl::set\_call\_state(*C++ function*), 85



GpiCbHdl::set\_user\_data (C++ function), 85  
 GpiClockHdl (C++ class), 53, 86  
 GpiClockHdl::~~GpiClockHdl (C++ function), 86  
 GpiClockHdl::GpiClockHdl (C++ function), 86  
 GpiClockHdl::start\_clock (C++ function), 86  
 GpiClockHdl::stop\_clock (C++ function), 86  
 GpiHdl (C++ class), 53, 83  
 GpiHdl::~~GpiHdl (C++ function), 83  
 GpiHdl::get\_handle (C++ function), 83  
 GpiHdl::gpi\_copy\_name (C++ function), 83  
 GpiHdl::GpiHdl (C++ function), 83, 84  
 GpiHdl::initialise (C++ function), 83  
 GpiHdl::is\_this\_impl (C++ function), 83  
 GpiHdl::m\_impl (C++ member), 83  
 GpiHdl::m\_obj\_hdl (C++ member), 83  
 GpiImplInterface (C++ class), 53, 87  
 GpiImplInterface::~~GpiImplInterface (C++ function), 87  
 GpiImplInterface::deregister\_callback (C++ function), 88  
 GpiImplInterface::get\_name\_c (C++ function), 87  
 GpiImplInterface::get\_name\_s (C++ function), 87  
 GpiImplInterface::get\_root\_handle (C++ function), 87  
 GpiImplInterface::get\_sim\_precision (C++ function), 87  
 GpiImplInterface::get\_sim\_time (C++ function), 87  
 GpiImplInterface::GpiImplInterface (C++ function), 87  
 GpiImplInterface::iterate\_handle (C++ function), 87  
 GpiImplInterface::m\_name (C++ member), 88  
 GpiImplInterface::native\_check\_create (C++ function), 87  
 GpiImplInterface::reason\_to\_string (C++ function), 88  
 GpiImplInterface::register\_nexttime\_callback (C++ function), 88  
 GpiImplInterface::register\_readonly\_callback (C++ function), 88  
 GpiImplInterface::register\_readwrite\_callback (C++ function), 88  
 GpiImplInterface::register\_timed\_callback (C++ function), 88  
 GpiImplInterface::sim\_end (C++ function), 87  
 GpiIterator (C++ class), 53, 86  
 GpiIterator::~~GpiIterator (C++ function), 87  
 GpiIterator::get\_parent (C++ function), 87  
 GpiIterator::GpiIterator (C++ function), 87  
 GpiIterator::m\_parent (C++ member), 87  
 GpiIterator::next\_handle (C++ function), 87  
 GpiIteratorMapping (C++ class), 53, 87  
 GpiIteratorMapping::add\_to\_options (C++ function), 87  
 GpiIteratorMapping::get\_options (C++ function), 87  
 GpiIteratorMapping::GpiIteratorMapping (C++ function), 87  
 GpiIteratorMapping::options\_map (C++ member), 87  
 GpiObjHdl (C++ class), 53, 84  
 GpiObjHdl::~~GpiObjHdl (C++ function), 84  
 GpiObjHdl::get\_const (C++ function), 84  
 GpiObjHdl::get\_definition\_file (C++ function), 84  
 GpiObjHdl::get\_definition\_name (C++ function), 84  
 GpiObjHdl::get\_fullname (C++ function), 84  
 GpiObjHdl::get\_fullname\_str (C++ function), 84  
 GpiObjHdl::get\_indexable (C++ function), 84  
 GpiObjHdl::get\_name (C++ function), 84  
 GpiObjHdl::get\_name\_str (C++ function), 84  
 GpiObjHdl::get\_num\_elems (C++ function), 84  
 GpiObjHdl::get\_range\_left (C++ function), 84  
 GpiObjHdl::get\_range\_right (C++ function), 84  
 GpiObjHdl::get\_type (C++ function), 84  
 GpiObjHdl::get\_type\_str (C++ function), 84  
 GpiObjHdl::GpiObjHdl (C++ function), 84  
 GpiObjHdl::initialise (C++ function), 84  
 GpiObjHdl::is\_native\_impl (C++ function), 84  
 GpiObjHdl::m\_const (C++ member), 85  
 GpiObjHdl::m\_definition\_file (C++ member), 85  
 GpiObjHdl::m\_definition\_name (C++ member), 84  
 GpiObjHdl::m\_fullname (C++ member), 84  
 GpiObjHdl::m\_indexable (C++ member), 84  
 GpiObjHdl::m\_name (C++ member), 84  
 GpiObjHdl::m\_num\_elems (C++ member), 84  
 GpiObjHdl::m\_range\_left (C++ member), 84  
 GpiObjHdl::m\_range\_right (C++ member), 84  
 GpiObjHdl::m\_type (C++ member), 85  
 GpiSignalObjHdl (C++ class), 54, 85  
 GpiSignalObjHdl::~~GpiSignalObjHdl (C++ function), 85  
 GpiSignalObjHdl::get\_signal\_value\_binstr (C++ function), 85  
 GpiSignalObjHdl::get\_signal\_value\_long (C++ function), 85  
 GpiSignalObjHdl::get\_signal\_value\_real (C++ function), 85  
 GpiSignalObjHdl::get\_signal\_value\_str (C++ function), 85

GpiSignalObjHdl::GpiSignalObjHdl (C++ function), 85  
 GpiSignalObjHdl::m\_length (C++ member), 85  
 GpiSignalObjHdl::set\_signal\_value (C++ function), 85  
 GpiSignalObjHdl::value\_change\_cb (C++ function), 85  
 GPITrigger (class in cocotb.triggers), 33  
 GpiValueCbHdl (C++ class), 54, 86  
 GpiValueCbHdl::~~GpiValueCbHdl (C++ function), 86  
 GpiValueCbHdl::cleanup\_callback (C++ function), 86  
 GpiValueCbHdl::GpiValueCbHdl (C++ function), 86  
 GpiValueCbHdl::m\_signal (C++ member), 86  
 GpiValueCbHdl::required\_value (C++ member), 86  
 GpiValueCbHdl::run\_callback (C++ function), 86  
 gtstate (C++ member), 81  
 GUI, 11

## H

handle\_fli\_callback (C++ function), 57  
 handle\_gpi\_callback (C++ function), 89  
 handle\_vhpi\_callback (C++ function), 69  
 handle\_vpi\_callback (C++ function), 75  
 hexdiffs () (in module cocotb.utils), 39  
 hexdump () (in module cocotb.utils), 39  
 HierarchyArrayObject (class in cocotb.handle), 41  
 HierarchyObject (class in cocotb.handle), 41  
 hook (class in cocotb), 28

## I

idle () (cocotb.drivers.xgmii.XGMII method), 47  
 in\_reset () (cocotb.monitors.BusMonitor property), 36  
 INITERROR (C macro), 88  
 integer () (cocotb.binary.BinaryValue property), 31  
 IntegerObject (class in cocotb.handle), 43  
 is\_const (C++ function), 69  
 is\_enum\_boolean (C++ function), 69  
 is\_enum\_char (C++ function), 69  
 is\_enum\_logic (C++ function), 69  
 is\_python\_context (C++ member), 80  
 is\_resolvable () (cocotb.binary.BinaryValue property), 31  
 iterate (C++ function), 89, 91

## J

Join (class in cocotb.triggers), 20

## K

kill () (cocotb.drivers.Driver method), 33

## L

layer1 () (cocotb.drivers.xgmii.XGMII static method), 47  
 layer\_entry\_func (C++ type), 82  
 lazy\_property (class in cocotb.utils), 40  
 loads () (cocotb.handle.NonConstantObject method), 42  
 local\_level (C++ member), 82  
 Lock (class in cocotb.triggers), 21  
 locked (cocotb.triggers.Lock attribute), 21  
 log\_buff (C++ member), 82  
 log\_level (C++ function), 82, 89, 91  
 log\_msg (C++ function), 89, 90  
 LOG\_SIZE (C macro), 82

## M

ModifiableObject (class in cocotb.handle), 42  
 MODULE\_ENTRY\_POINT (C macro), 88  
 MODULE\_ENTRY\_POINT (C++ function), 92  
 MODULE\_NAME (C macro), 90  
 module\_state (C++ class), 88, 92  
 module\_state::error (C++ member), 88  
 moduledef (C++ member), 92  
 Monitor (class in cocotb.monitors), 36

## N

n\_bits () (cocotb.binary.BinaryValue property), 31  
 NATIVE (C++ enumerator), 86  
 NATIVE\_NO\_NAME (C++ enumerator), 86  
 next (C++ function), 89, 91  
 NextTimeStep (class in cocotb.triggers), 20  
 NonConstantObject (class in cocotb.handle), 42  
 NonHierarchyIndexableObject (class in cocotb.handle), 41  
 NonHierarchyObject (class in cocotb.handle), 41  
 NOT\_NATIVE (C++ enumerator), 86  
 NOT\_NATIVE\_NO\_NAME (C++ enumerator), 86

## O

OneToMany (C++ enum), 65  
 OPBMaster (class in cocotb.drivers.opb), 46  
 OTM\_CONSTANTS (C++ enumerator), 65  
 OTM\_END (C++ enumerator), 65  
 OTM\_REGIONS (C++ enumerator), 65  
 OTM\_SIGNAL\_SUB\_ELEMENTS (C++ enumerator), 65  
 OTM\_SIGNALS (C++ enumerator), 65  
 OTM\_VARIABLE\_SUB\_ELEMENTS (C++ enumerator), 65

## P

[p\\_callback\\_data \(C++ type\)](#), 90  
[pack \(\) \(in module cocotb.utils\)](#), 39  
[ParametrizedSingleton \(class in cocotb.utils\)](#), 40  
[pEventFn \(C++ member\)](#), 81  
[pLogFilter \(C++ member\)](#), 82  
[pLogHandler \(C++ member\)](#), 82  
[prime \(\) \(cocotb.triggers.Trigger method\)](#), 32  
[progname \(C++ member\)](#), 81  
[Python Enhancement Proposals](#)  
[PEP 525](#), 18  
[PYTHON\\_INTERPRETER\\_PATH \(C++ member\)](#), 81

## R

[raise\\_error \(\) \(in module cocotb.result\)](#), 27  
[read \(\) \(cocotb.drivers.amba.AXI4LiteMaster method\)](#), 45  
[read \(\) \(cocotb.drivers.avalon.AvalonMaster method\)](#), 45  
[read \(\) \(cocotb.drivers.opb.OPBMaster method\)](#), 46  
[ReadOnly \(class in cocotb.triggers\)](#), 19  
[ReadWrite \(class in cocotb.triggers\)](#), 20  
[RealObject \(class in cocotb.handle\)](#), 42  
[RegionObject \(class in cocotb.handle\)](#), 41  
[register\\_embed \(C++ function\)](#), 57, 69, 75  
[register\\_final\\_callback \(C++ function\)](#), 57, 69, 75  
[register\\_initial\\_callback \(C++ function\)](#), 57, 69, 75  
[register\\_nextstep\\_callback \(C++ function\)](#), 89, 91  
[register\\_readonly\\_callback \(C++ function\)](#), 89, 91  
[register\\_rwsynch\\_callback \(C++ function\)](#), 89, 91  
[register\\_system\\_functions \(C++ function\)](#), 75  
[register\\_timed\\_callback \(C++ function\)](#), 89, 91  
[register\\_value\\_change\\_callback \(C++ function\)](#), 89, 91  
[registered\\_impls \(C++ member\)](#), 69  
[reject\\_remaining\\_kwargs \(\) \(in module cocotb.utils\)](#), 40  
[release \(\) \(cocotb.triggers.Lock method\)](#), 22  
[releases \(C++ member\)](#), 90  
[remove\\_traceback\\_frames \(\) \(in module cocotb.utils\)](#), 40  
[result \(\) \(cocotb.scoreboard.Scoreboard property\)](#), 37  
[ReturnValue](#), 27  
[retval \(\) \(cocotb.triggers.Join property\)](#), 21  
[RisingEdge \(class in cocotb.triggers\)](#), 19  
[rx\\_data\\_to\\_ad9361 \(\) \(cocotb.drivers.ad9361.AD9361 method\)](#), 44

## S

[s\\_callback\\_data \(C++ type\)](#), 90  
[sample \(\) \(cocotb.bus.Bus method\)](#), 32  
[sample \(\) \(cocotb.wavedrom.Wavedrom method\)](#), 48  
[Scoreboard \(class in cocotb.scoreboard\)](#), 36  
[send \(cocotb.drivers.Driver attribute\)](#), 33  
[send\\_data \(\) \(cocotb.drivers.ad9361.AD9361 method\)](#), 43  
[set \(\) \(cocotb.triggers.Event method\)](#), 21  
[set\\_log\\_filter \(C++ function\)](#), 82  
[set\\_log\\_handler \(C++ function\)](#), 82  
[set\\_log\\_level \(C++ function\)](#), 82  
[set\\_program\\_name\\_in\\_venv \(C++ function\)](#), 81  
[set\\_signal\\_val\\_long \(C++ function\)](#), 89, 91  
[set\\_signal\\_val\\_real \(C++ function\)](#), 89, 91  
[set\\_signal\\_val\\_str \(C++ function\)](#), 89, 91  
[set\\_valid\\_generator \(\) \(cocotb.drivers.ValidatedBusDriver method\)](#), 35  
[setimmediatevalue \(\) \(cocotb.handle.EnumObject method\)](#), 42  
[setimmediatevalue \(\) \(cocotb.handle.IntegerObject method\)](#), 43  
[setimmediatevalue \(\) \(cocotb.handle.ModifiableObject method\)](#), 42  
[setimmediatevalue \(\) \(cocotb.handle.RealObject method\)](#), 42  
[setimmediatevalue \(\) \(cocotb.handle.StringObject method\)](#), 43  
[signed\\_integer \(\) \(cocotb.binary.BinaryValue property\)](#), 31  
[SIGNED\\_MAGNITUDE \(cocotb.binary.BinaryRepresentation attribute\)](#), 30  
[SIM](#), 11  
[SIM\\_ARGS](#), 11  
[SIM\\_BUILD](#), 12  
[sim\\_finish\\_cb \(C++ member\)](#), 57, 70, 75  
[sim\\_init\\_cb \(C++ member\)](#), 57, 70, 75  
[sim\\_time \(C++ class\)](#), 90, 92  
[sim\\_time::high \(C++ member\)](#), 90  
[sim\\_time::low \(C++ member\)](#), 90  
[sim\\_to\\_hdl \(C++ function\)](#), 83  
[SimFailure](#), 27  
[SimHandle \(\) \(in module cocotb.handle\)](#), 43  
[SimHandleBase \(class in cocotb.handle\)](#), 41  
[simulator\\_clear \(C++ function\)](#), 92  
[simulator\\_traverse \(C++ function\)](#), 92  
[SimulatorMethods \(C++ member\)](#), 91  
[start \(cocotb.clock.Clock attribute\)](#), 38  
[start \(\) \(cocotb.drivers.BitDriver method\)](#), 34  
[Status \(C++ enum\)](#), 86  
[stop \(\) \(cocotb.drivers.BitDriver method\)](#), 34  
[stop\\_simulator \(C++ function\)](#), 89, 91



StringObject (class in cocotb.handle), 43  
 system\_function\_compiletf (C++ function), 75  
 system\_function\_overload (C++ function), 75  
 systf\_error\_level (C++ member), 75  
 systf\_fatal\_level (C++ member), 75  
 systf\_info\_level (C++ member), 75  
 systf\_warning\_level (C++ member), 75

## T

t\_callback\_data (C++ class), 91, 93  
 t\_callback\_data::\_saved\_thread\_state (C++ member), 91  
 t\_callback\_data::args (C++ member), 91  
 t\_callback\_data::cb\_hdl (C++ member), 92  
 t\_callback\_data::function (C++ member), 91  
 t\_callback\_data::id\_value (C++ member), 91  
 t\_callback\_data::kwargs (C++ member), 92  
 TAKE\_GIL (C++ function), 90  
 takes (C++ member), 90  
 terminate() (cocotb.drivers.xgmii.XGMII method), 47  
 test (class in cocotb), 28  
 TestComplete, 27  
 TestError, 27  
 TestFactory (class in cocotb.regression), 28  
 TestFailure, 27  
 TestSuccess, 27  
 Timer (class in cocotb.triggers), 19  
 to\_gpi\_objtype (C++ function), 75  
 to\_python (C++ function), 80  
 to\_simulator (C++ function), 80  
 trace (class in cocotb.wavedrom), 48  
 Trigger (class in cocotb.triggers), 32  
 TWOS\_COMPLEMENT (cocotb.binary.BinaryRepresentation attribute), 30  
 tx\_data\_from\_ad9361() (cocotb.drivers.ad9361.AD9361 method), 44

## U

unpack() (in module cocotb.utils), 39  
 unprime() (cocotb.triggers.GPITrigger method), 33  
 unprime() (cocotb.triggers.Trigger method), 33  
 UNSIGNED (cocotb.binary.BinaryRepresentation attribute), 30  
 utils\_dyn\_open (C++ function), 80  
 utils\_dyn\_sym (C++ function), 80

## V

ValidatedBusDriver (class in cocotb.drivers), 35  
 value() (cocotb.binary.BinaryValue property), 31  
 value() (cocotb.handle.NonHierarchyObject property), 41  
 VERILOG\_SOURCES, 11

VHDL\_SOURCES, 11  
 VHDL\_SOURCES\_lib, 11  
 vhpi\_mappings (C++ function), 69  
 vhpi\_startup\_routines (C++ member), 70  
 vhpi\_startup\_routines\_bootstrap (C++ function), 69  
 vhpi\_table (C++ member), 70  
 VHPI\_TYPE\_MIN (C macro), 69  
 VhpiArrayObjHdl (C++ class), 54, 72  
 VhpiArrayObjHdl::~~VhpiArrayObjHdl (C++ function), 72  
 VhpiArrayObjHdl::initialise (C++ function), 72  
 VhpiArrayObjHdl::VhpiArrayObjHdl (C++ function), 72  
 VhpiCbHdl (C++ class), 54, 70  
 VhpiCbHdl::~~VhpiCbHdl (C++ function), 70  
 VhpiCbHdl::arm\_callback (C++ function), 70  
 VhpiCbHdl::cb\_data (C++ member), 70  
 VhpiCbHdl::cleanup\_callback (C++ function), 70  
 VhpiCbHdl::vhpi\_time (C++ member), 70  
 VhpiCbHdl::VhpiCbHdl (C++ function), 70  
 VhpiImpl (C++ class), 54, 74  
 VhpiImpl::create\_gpi\_obj\_from\_handle (C++ function), 74  
 VhpiImpl::deregister\_callback (C++ function), 74  
 VhpiImpl::format\_to\_string (C++ function), 74  
 VhpiImpl::get\_root\_handle (C++ function), 74  
 VhpiImpl::get\_sim\_precision (C++ function), 74  
 VhpiImpl::get\_sim\_time (C++ function), 74  
 VhpiImpl::iterate\_handle (C++ function), 74  
 VhpiImpl::m\_next\_phase (C++ member), 74  
 VhpiImpl::m\_read\_only (C++ member), 74  
 VhpiImpl::m\_read\_write (C++ member), 74  
 VhpiImpl::native\_check\_create (C++ function), 74  
 VhpiImpl::reason\_to\_string (C++ function), 74  
 VhpiImpl::register\_nexttime\_callback (C++ function), 74  
 VhpiImpl::register\_readonly\_callback (C++ function), 74  
 VhpiImpl::register\_readwrite\_callback (C++ function), 74  
 VhpiImpl::register\_timed\_callback (C++ function), 74  
 VhpiImpl::sim\_end (C++ function), 74  
 VhpiImpl::VhpiImpl (C++ function), 74  
 VhpiIterator (C++ class), 54, 73  
 VhpiIterator::~~VhpiIterator (C++ function),

73  
 VhpiIterator::iterate\_over (C++ member),  
 74  
 VhpiIterator::m\_iter\_obj (C++ member), 73  
 VhpiIterator::m\_iterator (C++ member), 73  
 VhpiIterator::next\_handle (C++ function), 73  
 VhpiIterator::one2many (C++ member), 73  
 VhpiIterator::selected (C++ member), 73  
 VhpiIterator::VhpiIterator (C++ function),  
 73  
 VhpiLogicSignalObjHdl (C++ class), 54, 73  
 VhpiLogicSignalObjHdl::~~VhpiLogicSignalObjHdl  
 (C++ function), 73  
 VhpiLogicSignalObjHdl::initialise (C++  
 function), 73  
 VhpiLogicSignalObjHdl::set\_signal\_value  
 (C++ function), 73  
 VhpiLogicSignalObjHdl::VhpiLogicSignalObjHdl  
 (C++ function), 73  
 VhpiNextPhaseCbHdl (C++ class), 54, 71  
 VhpiNextPhaseCbHdl::~~VhpiNextPhaseCbHdl  
 (C++ function), 71  
 VhpiNextPhaseCbHdl::VhpiNextPhaseCbHdl  
 (C++ function), 71  
 VhpiObjHdl (C++ class), 54, 72  
 VhpiObjHdl::~~VhpiObjHdl (C++ function), 72  
 VhpiObjHdl::initialise (C++ function), 72  
 VhpiObjHdl::VhpiObjHdl (C++ function), 72  
 VhpiReadOnlyCbHdl (C++ class), 54, 71  
 VhpiReadOnlyCbHdl::~~VhpiReadOnlyCbHdl  
 (C++ function), 71  
 VhpiReadOnlyCbHdl::VhpiReadOnlyCbHdl  
 (C++ function), 71  
 VhpiReadwriteCbHdl (C++ class), 54, 72  
 VhpiReadwriteCbHdl::~~VhpiReadwriteCbHdl  
 (C++ function), 72  
 VhpiReadwriteCbHdl::VhpiReadwriteCbHdl  
 (C++ function), 72  
 VhpiShutdownCbHdl (C++ class), 55, 71  
 VhpiShutdownCbHdl::~~VhpiShutdownCbHdl  
 (C++ function), 72  
 VhpiShutdownCbHdl::cleanup\_callback  
 (C++ function), 72  
 VhpiShutdownCbHdl::run\_callback (C++  
 function), 72  
 VhpiShutdownCbHdl::VhpiShutdownCbHdl  
 (C++ function), 72  
 VhpiSignalObjHdl (C++ class), 55, 72  
 VhpiSignalObjHdl::~~VhpiSignalObjHdl  
 (C++ function), 72  
 VhpiSignalObjHdl::chr2vhpi (C++ function),  
 73  
 VhpiSignalObjHdl::get\_signal\_value\_binstr  
 (C++ function), 72  
 VhpiSignalObjHdl::get\_signal\_value\_long  
 (C++ function), 72  
 VhpiSignalObjHdl::get\_signal\_value\_real  
 (C++ function), 72  
 VhpiSignalObjHdl::get\_signal\_value\_str  
 (C++ function), 72  
 VhpiSignalObjHdl::initialise (C++ func-  
 tion), 73  
 VhpiSignalObjHdl::m\_binvalue (C++ mem-  
 ber), 73  
 VhpiSignalObjHdl::m\_either\_cb (C++ mem-  
 ber), 73  
 VhpiSignalObjHdl::m\_falling\_cb (C++ mem-  
 ber), 73  
 VhpiSignalObjHdl::m\_rising\_cb (C++ mem-  
 ber), 73  
 VhpiSignalObjHdl::m\_value (C++ member), 73  
 VhpiSignalObjHdl::set\_signal\_value (C++  
 function), 72, 73  
 VhpiSignalObjHdl::value\_change\_cb (C++  
 function), 73  
 VhpiSignalObjHdl::VhpiSignalObjHdl (C++  
 function), 72  
 VhpiStartupCbHdl (C++ class), 55, 71  
 VhpiStartupCbHdl::~~VhpiStartupCbHdl  
 (C++ function), 71  
 VhpiStartupCbHdl::cleanup\_callback (C++  
 function), 71  
 VhpiStartupCbHdl::run\_callback (C++ func-  
 tion), 71  
 VhpiStartupCbHdl::VhpiStartupCbHdl (C++  
 function), 71  
 VhpiTimedCbHdl (C++ class), 55, 71  
 VhpiTimedCbHdl::~~VhpiTimedCbHdl (C++  
 function), 71  
 VhpiTimedCbHdl::cleanup\_callback (C++  
 function), 71  
 VhpiTimedCbHdl::VhpiTimedCbHdl (C++ func-  
 tion), 71  
 VhpiValueCbHdl (C++ class), 55, 70  
 VhpiValueCbHdl::~~VhpiValueCbHdl (C++  
 function), 70  
 VhpiValueCbHdl::cleanup\_callback (C++  
 function), 70  
 VhpiValueCbHdl::falling (C++ member), 71  
 VhpiValueCbHdl::initial\_value (C++ mem-  
 ber), 71  
 VhpiValueCbHdl::rising (C++ member), 71  
 VhpiValueCbHdl::signal (C++ member), 71  
 VhpiValueCbHdl::VhpiValueCbHdl (C++ func-  
 tion), 70  
 vlog\_startup\_routines (C++ member), 75  
 vlog\_startup\_routines\_bootstrap (C++  
 function), 75

vpi\_mappings (C++ function), 75  
 vpi\_table (C++ member), 75  
 VPI\_TYPE\_MAX (C macro), 74  
 VpiArrayObjHdl (C++ class), 55, 77  
 VpiArrayObjHdl::~~VpiArrayObjHdl (C++ function), 78  
 VpiArrayObjHdl::initialise (C++ function), 78  
 VpiArrayObjHdl::VpiArrayObjHdl (C++ function), 78  
 VpiCbHdl (C++ class), 55, 76  
 VpiCbHdl::~~VpiCbHdl (C++ function), 76  
 VpiCbHdl::arm\_callback (C++ function), 76  
 VpiCbHdl::cb\_data (C++ member), 76  
 VpiCbHdl::cleanup\_callback (C++ function), 76  
 VpiCbHdl::vpi\_time (C++ member), 76  
 VpiCbHdl::VpiCbHdl (C++ function), 76  
 VpiImpl (C++ class), 55, 79  
 VpiImpl::create\_gpi\_obj\_from\_handle (C++ function), 80  
 VpiImpl::deregister\_callback (C++ function), 80  
 VpiImpl::get\_root\_handle (C++ function), 79  
 VpiImpl::get\_sim\_precision (C++ function), 79  
 VpiImpl::get\_sim\_time (C++ function), 79  
 VpiImpl::iterate\_handle (C++ function), 79  
 VpiImpl::m\_next\_phase (C++ member), 80  
 VpiImpl::m\_read\_only (C++ member), 80  
 VpiImpl::m\_read\_write (C++ member), 80  
 VpiImpl::native\_check\_create (C++ function), 80  
 VpiImpl::next\_handle (C++ function), 79  
 VpiImpl::reason\_to\_string (C++ function), 80  
 VpiImpl::register\_nexttime\_callback (C++ function), 80  
 VpiImpl::register\_readonly\_callback (C++ function), 79  
 VpiImpl::register\_readwrite\_callback (C++ function), 80  
 VpiImpl::register\_timed\_callback (C++ function), 79  
 VpiImpl::sim\_end (C++ function), 79  
 VpiImpl::VpiImpl (C++ function), 79  
 VpiIterator (C++ class), 55, 78  
 VpiIterator::~~VpiIterator (C++ function), 79  
 VpiIterator::iterate\_over (C++ member), 79  
 VpiIterator::m\_iterator (C++ member), 79  
 VpiIterator::next\_handle (C++ function), 79  
 VpiIterator::one2many (C++ member), 79  
 VpiIterator::selected (C++ member), 79  
 VpiIterator::VpiIterator (C++ function), 79  
 VpiNextPhaseCbHdl (C++ class), 55, 77  
 VpiNextPhaseCbHdl::~~VpiNextPhaseCbHdl (C++ function), 77  
 VpiNextPhaseCbHdl::VpiNextPhaseCbHdl (C++ function), 77  
 VpiObjHdl (C++ class), 55, 78  
 VpiObjHdl::~~VpiObjHdl (C++ function), 78  
 VpiObjHdl::initialise (C++ function), 78  
 VpiObjHdl::VpiObjHdl (C++ function), 78  
 VpiReadOnlyCbHdl (C++ class), 56, 76  
 VpiReadOnlyCbHdl::~~VpiReadOnlyCbHdl (C++ function), 77  
 VpiReadOnlyCbHdl::VpiReadOnlyCbHdl (C++ function), 77  
 VpiReadwriteCbHdl (C++ class), 56, 77  
 VpiReadwriteCbHdl::~~VpiReadwriteCbHdl (C++ function), 77  
 VpiReadwriteCbHdl::VpiReadwriteCbHdl (C++ function), 77  
 VpiShutdownCbHdl (C++ class), 56, 77  
 VpiShutdownCbHdl::~~VpiShutdownCbHdl (C++ function), 77  
 VpiShutdownCbHdl::cleanup\_callback (C++ function), 77  
 VpiShutdownCbHdl::run\_callback (C++ function), 77  
 VpiShutdownCbHdl::VpiShutdownCbHdl (C++ function), 77  
 VpiSignalObjHdl (C++ class), 56, 78  
 VpiSignalObjHdl::~~VpiSignalObjHdl (C++ function), 78  
 VpiSignalObjHdl::get\_signal\_value\_binstr (C++ function), 78  
 VpiSignalObjHdl::get\_signal\_value\_long (C++ function), 78  
 VpiSignalObjHdl::get\_signal\_value\_real (C++ function), 78  
 VpiSignalObjHdl::get\_signal\_value\_str (C++ function), 78  
 VpiSignalObjHdl::initialise (C++ function), 78  
 VpiSignalObjHdl::m\_either\_cb (C++ member), 78  
 VpiSignalObjHdl::m\_falling\_cb (C++ member), 78  
 VpiSignalObjHdl::m\_rising\_cb (C++ member), 78  
 VpiSignalObjHdl::set\_signal\_value (C++ function), 78  
 VpiSignalObjHdl::value\_change\_cb (C++ function), 78  
 VpiSignalObjHdl::VpiSignalObjHdl (C++ function), 78  
 VpiSingleIterator (C++ class), 56, 79  
 VpiSingleIterator::~~VpiSingleIterator

(C++ *function*), [79](#)  
VpiSingleIterator::m\_iterator (C++ *member*), [79](#)  
VpiSingleIterator::next\_handle (C++ *function*), [79](#)  
VpiSingleIterator::VpiSingleIterator  
(C++ *function*), [79](#)  
VpiStartupCbHdl (C++ *class*), [56](#), [77](#)  
VpiStartupCbHdl::~~VpiStartupCbHdl (C++  
*function*), [77](#)  
VpiStartupCbHdl::cleanup\_callback (C++  
*function*), [77](#)  
VpiStartupCbHdl::run\_callback (C++ *function*), [77](#)  
VpiStartupCbHdl::VpiStartupCbHdl (C++  
*function*), [77](#)  
VpiTimedCbHdl (C++ *class*), [56](#), [76](#)  
VpiTimedCbHdl::~~VpiTimedCbHdl (C++ *function*), [76](#)  
VpiTimedCbHdl::cleanup\_callback (C++  
*function*), [76](#)  
VpiTimedCbHdl::VpiTimedCbHdl (C++ *function*), [76](#)  
VpiValueCbHdl (C++ *class*), [56](#), [76](#)  
VpiValueCbHdl::~~VpiValueCbHdl (C++ *function*), [76](#)  
VpiValueCbHdl::cleanup\_callback (C++  
*function*), [76](#)  
VpiValueCbHdl::m\_vpi\_value (C++ *member*),  
[76](#)  
VpiValueCbHdl::VpiValueCbHdl (C++ *function*), [76](#)

## W

wait() (*cocotb.triggers.Event method*), [21](#)  
wait\_for\_recv() (*cocotb.monitors.Monitor  
method*), [36](#)  
want\_color\_output() (*in module cocotb.utils*), [40](#)  
Wavedrom (*class in cocotb.wavedrom*), [48](#)  
WAVES, [11](#)  
write() (*cocotb.drivers.amba.AXI4LiteMaster  
method*), [44](#)  
write() (*cocotb.drivers.avalon.AvalonMaster  
method*), [45](#)  
write() (*cocotb.drivers.opb.OPBMaster method*), [46](#)

## X

XGMII (*class in cocotb.drivers.xgmii*), [47](#)  
XGMII (*class in cocotb.monitors.xgmii*), [48](#)