Teal User’s Manual

Version 0.60

Mike Mintz
Apple Valley Software

Table of Contents

List of Figures

About this Manual

This document describes Teal, a c/c++ Test Environment Abstraction Layer. It is intended for verification engineers who use Teal.

This manual assumes you are familiar with Hardware Description Languages (HDL) such as Verilog or VHDL. It also assumes a general knowledge of the problem of hardware verification and assumes you are familiar with software programming in c++. It also assumes you know the use of the Linux Operating System.

One-liner about what the document is about

Throughout this document, text written in `this font` is intended to be written in c/c++ or an HDL.

Customer support:

To report an error in Teal, go to www.sourceforge.com/projects/teal and click on the "bugs" section.

For support, send e-mail to support@applevalleysoftware.com.

Chapter 1 Overview

Engineers perform verification in order to:

- Minimize the probability of hardware functional errors
- Ensure that the hardware meets performance requirements
- Ensure that the hardware is usable by software

Teal aids in this endevor by providing a set of capabilities that access HDL signals and enable actions based on changes in the values of these signals. In addition, it encourages independent generators, transactors and checkers by providing for management of independent user created threads.

Because Teal is a c/c++ library, algorithms can be developed that both validate the hardware design and can be re-used in the production software. In addition, the c/c++ language is well defined and is well documented.

Chapter 2 Components of a Teal Verification System

The basic Teal verification components are a Teal library, a set of Verilog design files, a design top, and a set of c/c++ source files. There is also probably a run script and some makefiles.

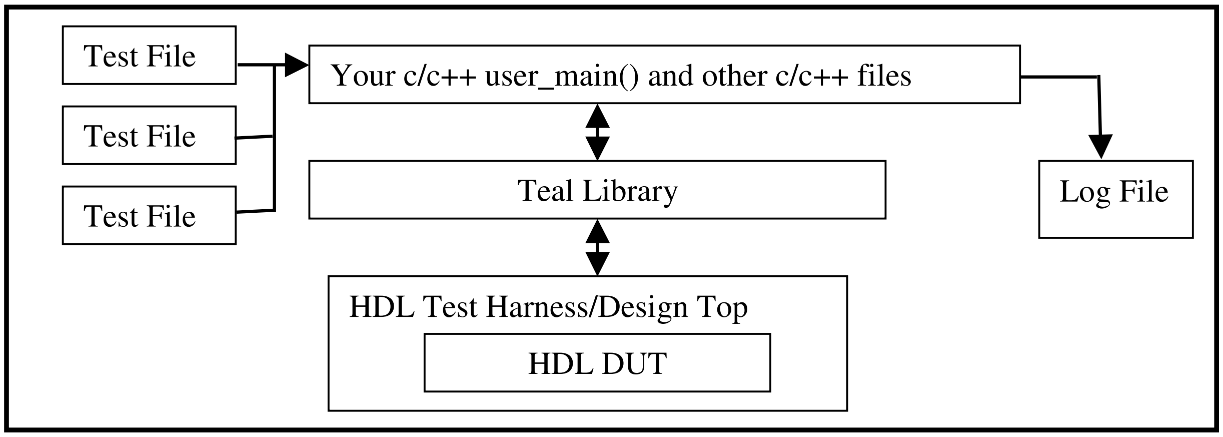

Test File
Test File
Test File
Your c/c++ user_main() and other c/c++ files
Teal Library
Log File
HDL Test Harness/Design Top
HDL DUT

Figure 1 Basic Components of Teal based verification

Chapter 3 Installation of Teal

To install Teal:

1. Go to from www.sourceforge.com/projects/teal.
2. Click the teal-binary download text.
3. Extract Teal to a directory, making sure that the directory you use is on your INCLUDE path (to enable C/C++ compilations).
4. Set the environment variable VERIL_INST_DIR to the path for your simulator
5. Set the environment variable SIM to your simulator (currently only ivl, mti, or mti_2_0)
6. `cd` to the test sub-directory and type "`run -c -clean $(SIM) -l test_list`" to run all the examples, or "`run -c -$SIM -t <some_test>`" to run a single test (denoted by a pair of .cpp and Verilog files with the same root name).

Chapter 4 The User Main Program

Section 4.1.1 Overview

This chapter discusses how Teal is hooked into the simulator. Specifically, this chapter describes what you must add to your HDL testbench code to use Teal.

Section 4.1.2 Theory of Operation

Teal uses PLI to allow c/c++ code to co-exist with the HDL. To this end, a call must be put somewhere in the HDL to start Teal. This is usually done in an initial block in the top level.

The call to `$teal_main` causes teal to start the threading system and call your `user_main()` function. Your `user_main()` function usually initializes global objects, like the random number generator, dictionary, and your own top level code and then waits for a init_done or out_of_reset signal from the HDL testbench.

After the DUT is ready, the `user_main()` usually starts a series of transactors and checkers, possible under the control of dictionary variables or command line switches. During execution, your code prints log messages for checks that pass and errors that occur.

The `user_main()` then waits for some signal from the lower level units that they are done. This may happen when a certain amount time has passed, or traffic generators are done, or maybe an error threshold has been reached.

Finally, the `user_main()` usually prints some statistics and signals to the HDL testbench to quit. The HDL testbench calls $finish to shut down the simulation.

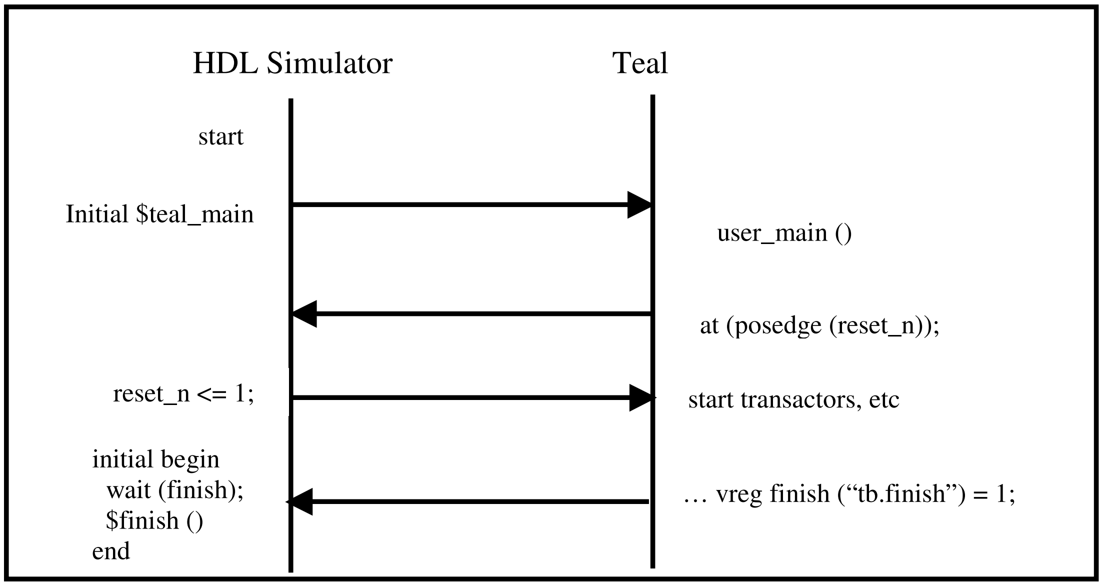
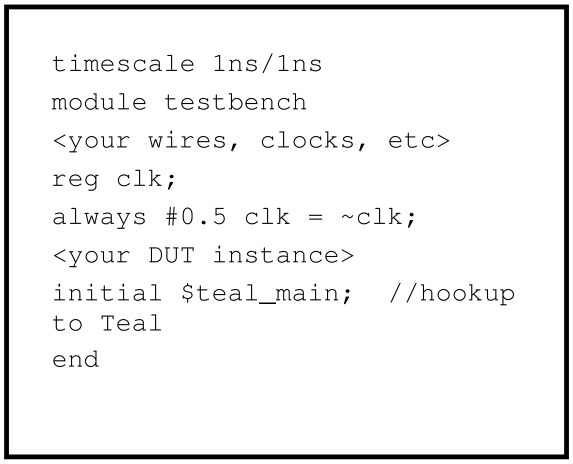


Figure 2 Process flow

Section 4.1.3 Interface

```
void user_main ();
```

Teal calls this function during initialization.



```
timescale 1ns/1ns
module testbench
<your wires, clocks, etc>
reg clk;
always #0.5 clk = ~clk;
<your DUT instance>
initial $teal_main;  //hookup
to Teal
end
```

Figure 3 An example Verilog testbench.v

Section 4.1.4 Examples

Given the testbench example above, this simple program just runs for 20 clock pulses.

```
#include “teal.h”
using namespace teal;
int user_main ()
{
  vreg clock (“testbench.clk”);
dictionary::start (“simple_clock_test.txt”);
uint number_of_periods (dictionary::find (“number_of_clocks”, 20));
dictionary::stop ();
for (int i(0); i < number_of_periods; ++i) {
  vout::get (note) << “i is “ << i << clock is << clock << endl;
}
vout::get (expected) << “test completed” << endl;
}
```

In this example, the test simply runs for a number of positive edges in a clock register. The duration is picked up by the test file “simple_clock_test.txt”. This file only has one line “number_of_clocks 234”, which defines the length of the run.

Chapter 5 The reg object

Section 5.1 Overview

When one starts to design a class library, an important decision is the creation of the "common currency" of the system. This chapter and the next few describe the common currency of the Teal system. These are the basic generic building blocks of a Teal based verification system. The reg clas is the most basic and is described first.

Section 5.1.1 Theory of Operation

Hardware simulators compute in four possible values for a bit[1], those being 1, 0, X, and Z. In addition, hardware languages support arrays of bits. The Teal class `reg` implements this four-state logic and makes sure X's propagate through calculations.

The `reg` class supports the usual HDL wire/register operations such as addition, subtraction, shifting, boolean operations, and four state comparison. Note that since multiplication and division are not part of the standard HDL operations, they are not provided. As in HDL languages, bitfields or subranges of `reg/vreg` is supported. They can be on either the left or right side of an expression.

The reg class supports display as an aval/bval series[2] or as a formatted hex string.

Section 5.2 C\C++ Interface

Section 5.2.1 Creating, copying and destroying a reg

There are four basic constructors and a copy constructor for `Reg`. The default constructor is marked explicit to prevent accindentially creating a one-bit register. The `length` class is a little helper class used in the constructor to force the register to be a specific length, for example, `reg a (length (45));` will create a 45 bit register. The length class only has an explicit integer constructor.

The `reg_slice` class is another helper class that is automatically created (and destroyed) when a register subfield is used. The register constructor is automatically used when a register is needed in an expression. For `example reg a(length(45)); reg b (a(10,0))`; will create a `reg_slice` of `a`. As `reg_slice` is a temporary object automatically created during left-hand assignment, it is not documented further.

[1] Some simulators use 8 logic vlaues, but Teal uses only four because the concept of drive strenght is not relevant to functional verification.

[2] As per the standard, i.e if bval is 0, aval is the usual 0 or 1, or is bval is 1 aval is X or Z, respectively

The last reg constructor takes in an integer and creates a 64-bit reg set to that value. This constructor allows statements like `reg a(length(32)); a = a + 5;` and `a(4:0) += 1;`.

The operator=() method sets `this` to the rhs but does not change the length of this. Specifically, it will extend (with 0) or truncate, as appropriate, depending on the length of the rhs.

The virtual destructor allows for subclasses to cleanup any allocated storage.

```
reg () [explicit]
reg (const length &)
reg (const reg_slice &)
reg (uint64)
reg (const reg &)
reg& reg::operator= (const reg &)
~reg () [virtual]
```

Section 5.2.2 Reg access functions

This set of functions provides read/write/display capability.

The single argument operator() is the bit index function of a register. You use this function to get a single bit. You must use the two argument function to set a single bit.

The constant two argument function returns a copy of the bits of the reg as specified by the arguments. This is called automatically by the compiler when the slicing operation is on the right hand side of an expression.

The two argument non-constant function returns a reference to the bits of the reg as specified by the arguments. This is called automatically by the compiler when the slicing operation is on the left hand side of an expression.

```
char operator() (uint32 b) const
reg operator() (uint32 u, uint32 l) const
reg_slice operator() (uint32 u, uint32 l)
```

Section 5.2.3 Math functions

These functions come in two flavors. One is as a global function. The other is as a member function that sends its results back to itself (the <x>= operators). The global functions exist because they are symmetrical. They are used when it is not appropriate to put it as a member function, usually because to reg objects are passed in or because the operation is commutative.

```
reg operator+ (const reg & lhs, const reg & rhs)
reg operator- (const reg & lhs, const reg & rhs)
reg& operator+= (const reg & rhs)
reg& operator-= (const reg & rhs)
```

The functions listed below perform left and right shift operations. There are equivalent symmetrical operators as well. These functions, while seemingly complex, provide capabilities that make `reg` act like a built-in type.

Note that 0's are shifted in during a right shift.

```
reg& roperator<< (unsigned rhs)
reg& operator>> (unsigned[3] rhs)
vout& operator<< (vout & c, const reg & rhs)
reg operator>> (const reg & lhs, const uint32 rhs)
```

Section 5.2.4 Logic functions

These functions implement the boolean operations associated with a register. Some functions also implement the logic as a four_state enumeration as in HDL. As with the math functions, some are global and symmetrical, while others are methods. They are used when it is not appropriate to put it as a member function, usually because to reg objects are passed in or because the operation is commutative.

By default, the relational (less or greater than) operators act like the normal two-state mathematical less-than. However, when a single bit is X, the result is X.

The triple_equal function is intended to model the "===" operator of Verilog.

```
enum four_state {zero=0, one, X, Z}
bool operator== (const reg & lhs, const reg & rhs)
reg operator~ (const reg & lhs)
reg operator| (const reg & lhs, const reg & rhs)
reg& reg::operator|= (const reg & rhs)
reg operator & (const reg & lhs, const reg & rhs)
reg& reg::operator &= (const reg & rhs)
bool operator!= (const reg & lhs, const reg & rhs)
four_state operator< (const reg & lhs, const reg & rhs)
four_state operator< (const reg & lhs, const reg & rhs)
four_state triple_equal (const reg & lhs, const reg & rhs)
```

[3] Shouldn't this take a uint32?

Section 5.2.5 Printing functions

The following functions allow a reg (and any derived classes) to be printed as part of a vout line. See vout, in Chapter 7. The reg::operator<<() is the virtual method to allow reg and all subclasses to be printed as a built-in type. By default, it prints the aval/bval state of the object and its bit length.[4]

The format_string() function returns a hex representation of the reg, e.g. 4'hf.[5]

```
vout & reg::operator<< (vout & c) const [virtual]
vout & operator<< (vout & c, const reg&)
std::string format_string () const
```

Section 5.2.6 Functions for HDL coherency

The following methods are used by derived classes like vreg to allow it to make sure the HDL signal and the c/c++ world agrees.Within reg, all these methods do nothing. The read_check() method is called before every access to the reg internal storage (aval/bval array). The write_check() method is the corollary to read_check(), in that it is called whenever any part of the reg is updated.

```
virtual void read_check () const [virtual]
virtual void write_through () const [virtual]
```

Section 5.3 Examples:

This section shows some basic examples of reg and vreg. They can be used as an introduction of the capabilities and use of these classes.

1. Declarations:

```
reg a(23);  //64 bit register/initialize it with the value 23
reg b(length (323)); //323 bit register with the initial value of all X's
b = 0x11; //assign b to 11 (clearing the upper bits)
b(315,300) = a; //set bits 315 to 300 of b to 23.
a = b(63,0); //uses the lower 64 bits of b
reg c(b); //323 it register with the initial value of b's current value
c(440, 413) = 1;  //illegal - run time error, bit index out of range
vreg d("tb.chip.reset_n); //create a vreg that is tied to reset_n
c = d; //clear all but the lowest bit of c, c(0) = current value of
reset_n
```

[4] Note to me. How to cleanly hook this into format_string?

[5] Not to me; Should really get the vout mode and format it to hex or decimal. What about a format_int? or at lest making the operator<< out put an int. Isn't 99% <= 64 bits?

```
d = 0x0; //push reset_n to 0;
```

2. Math on reg/vreg.

```
std::string path (“testbench.top.main_bus”); //the root of the bus module
vreg addr(path + “.address”); //address, bit length copied from HDL
reg b(length (32));
b(31:0) = 0x12; //init b
addr += 2; //increment address, push to HDL
addr = b << 3; //same as addr(addr.length() -1, 3) = b; addr(2,0) = 0;
```

3. Bit fields

```
reg b (101);
std::string root (“tb”);
std::string module (“bus”);
vreg addr (root + “.” + module + “.rd_addr”);
addr(1,0) = b(28,27);  //copy the two bits, push to HDL
int c (addr (31,28).format_int()); //get current upper nibble of address
addr(27,20) &= 0x55; //do some bit bashing
```

Chapter 6 The vreg object

Section 6.1 Overview

The reg class is neat, but, by itself, has very little to do with simulation. The vreg class is derived from reg, and implements the hookup to the HDL.

Section 6.2 Theory of Operation

Passing in a path to the HDL creates a vreg, e.g. `vreg ("testbench.uart.clk")` or `vreg ("tb.usb_dp").` `Vreg`'s differ from `regs` in that their value is initilized and conceptually exists in the HDL. Therefore, any assignment (or subrange assignment) is pushed to the HDL as if it were a non-blocking assignment[6][7].

Because reg was designed with vreg in mind, there are backdoor methods to know when a reg should be written to the HDL or the most recent HDL value is needed.

Section 6.3 C/C++ Interface

Section 6.3.1 Creating, copying, and destroying a vreg

The way to create a vreg is to pass in a string path down to the wire or register. The path is copied so that later printing functions can print the signal by name. If the path does not map to an HDL signal, an error is issued an all subsequent usage of this vreg will fail.

```
vreg (const std::string & path_and_name)
~vreg ()
vreg & vreg::operator= (const reg &)[8]
```

Section 6.3.2 Functions for HDL coherency

The following methods are overridden by vreg to allow it to make sure the HDL signal and the c/c++ world agrees. The read_check() method checks the current integer global state and if it's value is not the same, calls the HDL to get the current value of the signal. Then it sets its internal state variable to the global one. The write_check() implementation pushes a value to the HDL, as though it were a non blocking assignment.

[6] Note to me: Write a ctor from a string and consider adding rand method or additional function.

[7] Note to me: Implement a 2 state reg?

[8] This is a bug in reg::operator=() It must call the write_through() at the end.

Invalidate_all_vregs() is a method used only by the threads management system. It is called when the c/c++ code is called from the HDL side. This causes the global state value to change, causing all vregs to get a new value when/if they are referenced.

```
virtual void read_check () const
void write_through () const
void vinvalidate_all_vregs () [static]
```

Section 6.4 Examples:

This section shows some basic examples of reg and vreg. They can be used as an introduction of the capabilities and use of these classes.

1. Declarations:

```
std::string path (“testbench.top.main_bus”); //root of the main bus
vreg addr(path + “.address”); //access the address, bit length from HDL
vreg clk((“testbench.top.clk”);
vreg a(23);  //illegal. Need a path
reg b(length (72)); //create a 72 bit register, initial value of all X’s
b(71,32) = addr; //set bits 71 to 32 of b to current value of address.
clk = 0x0; //force clk to 0;
```

2. Bit fields

```
reg b (101);
std::string root (“tb”);
std::string module (“bus”);
vreg addr (root + “.” + module + “.rd_addr”);
addr(1,0) = b(28,27);  //copy the two bits, push to HDL
int c (addr (31,28).format_int()); //get current upper nibble of address
addr(27,20) &= 0x55; //do some bit bashing
```

Chapter 7 Logging simulation output

Section 7.1 Theory of Operation

Often, a log file is used as a trace of what happened during a simulation. It is important to have a consistent message format, to enable post processing, error counting and possibly filtering[9]. The Teal class `vlog` encourages such uniformity.

The vlog class is modeled after the c++ cout object. It directly supports output of the standard types. By following a few simple rules, complex objects can be printed as conviently as the standard types. See example three.

The vout class supports the concept of a singleton. That is, there is only one logger in the entire system and it is globally available. The method `vout::get (message_type)` creates the vout object if it does not exist and starts a line of the appropriate message type. The message type can be either `note`, `expected` or `error`. The `note` type is used for standard messages. The `expected` type is used when a test has been performed and the result is correct. An example would be receiving the correct byte at the correct time. The `error` type is used when the test failed. The number of expected and error messages are counted and can be used for end of test checking or summary reports.

When the vout object is created, it scans the command line for `+output_file+<filename>.` If found, all output is also sent to the file specified.[10]

The vout class also supports either decimal or hex output. By placing either a `hex` or `dec` in the vout message line, that output basis is selected.

Section 7.2 C/C++ Interface

Section 7.2.1 Creating, copying, and destroying a Vout

Vout is unlike a "normal" class in that it is expected to always be there. As such, there is no explicit construction step. By simple calling either of the get() methods, a vout will be created. Also, because vout is implementing a singleton, it cannot be copied or assigned. Finally, because it will exist for the duration of the simulation, it is not destroyed.

The message type describes whaqt type of message is being printed.

[9] Note to me: Add a mechinism to override what type of vour gets created, make all outupt methods virtual. Conside an add on package to filter output. Consider printing the thread name. Consider verilog hook to capture verilog output (like teal_display()). Could get messy though. Consider adding a error threshold, after which the system would be shutdown.

[10] Note to me: Have to move this scan args into a arg_dictionary object, or maybe merge it with the dictionary, maybe with a call in the dictionary namespace.

```
enum message_type = {
note
expected
error}
vout & get () [static]
vout & get (message_type) [static]
```

Section 7.2.2 Statistics

As each type of message is printed, vout increments a counter. The current values of the counters are retrived by the following calls.

```
int error_count () [static]
int expected_count () [static]
int note_count () [static]
```

Section 7.2.3 Output of the standard types

The following functions handle outputting of the standard types in c/c++.

```
vout& operator<< (vout &(* f)(vout &))
vout & operator<< (double)
vout & operator<< (const std::string &)
vout & operator<< (long long unsigned int)
vout & operator<< (long)
vout & operator<< (unsigned int)
vout & operator<< (int)
vout & operator<< (char)
```

Section 7.2.4 Output of the standard types

The following functions set the output type and end a line. WARNING: You must call `endl()`to end a line. This is the matching call to the `get(message_type)` function. Calling `endl()` signals to the vout object to allow another thread to begin output.

```
vout& dec (vout & a_vout)
vout& endl (vout & a_vout)
vout& hex (vout & a_vout)
```

Section 7.3 Examples:

1: Simple

```
#include "vout.h"
```

```
using namespace teal;
Vout::get(note) << “Hello World ” << 42 << endl;
```

Assuming this is called at sim time 565 nano-seconds, the output is:

[565 ns] Hello World 42

2. Logging a reg

```
reg a(23);
reg b(length (48)); b[7:4] = 3;
vout::get(note) << “basic a is” << a << endl;
vout::get(expected) << “a is” << a << “ and also 0x“ << hex << a << endl;
vout::get (error) <, “b is “ << b.format_string () << endl;
```

Assuming this is called at sim time 565 nano-seconds, the output is:

[565 ns] Note: basic a is bit length 64, aval[0] = 23, aval[1] 0, bval[0] = 0, bval[1] = 0

[565 ns] EXPECTED: a is 23 and also 0x17

[565 ns] ERROR: b is 512'hXXXXXXXXXXX3X

3. Output of an object:

In this example, code will be shown to enable complex objects to be printed as native types. First, Declare the base operator<<() method as virtual, to allow derived classes to have their own output.

```
#include <vout.h> using namespace teal;
class base {
  virtual vout& operator<< (vout& v)  const {v << “Base class”; return
v;}
};
```

Now, declare a single global function to call the virtual one. The compiler will automatically find this function when you do a `vout::get(note) << base_instance << endl;`

```
inline vout& operator<< (vout& v, const base& b) {return b.operator<<
(v);}
class derived : public base {
  virtual vout& operator<< (vout& v) const { v << “Derived “; return v;}
};
base a_base;
derived a_derived;
vout::get (note) << a_base << “ and “ << a_derived << endl;
```

Assuming this is called at sim time 565 nano-seconds, the output is:

[565 ns] Base class and derived

Chapter 8 Random Number Generation

Section 8.1 Overview

Using random numbers for test values is a staple of modern verification. The numbers must be well distributed and be stable across runs. The first is important because you need to find all the possible valid values and the second is important because, once your found a bug, you will need to rerun that simulation at least twice: once to create a vcd file and another to confirm its fixed. You might want to put that run in a regression suite as well. The re-runs of that simulation must produce exactly the same sequence of random numbers.

This chapter describes Teal's random number capability.

Section 8.2 Theory of operation

Teal's trandom class provides a simple stable random number generator. In order to provide independent streams of random numbers, it takes in a string and in integer and uses this to create the start seed[11]. In addition, the random class is initialized with a master seed, which provides a tie in to all the streams. In this way, a single number, given to the static `random::init()` function, possibly from the command line[12] or test file, can be used to guide all random number streams.

The basic random number class generates a [0..1.0) random double on every draw. The `random_range` class maps this double to a range of integers. The base object is simple to allow for subclasses that are more complicated.

In order to support stability, the string and or number passed into the constructor must be carefully chosen. The examples below show some common techniques that have been used extensively to provide the appropriate level of flexibility and stability.

[11] Note to me: Vera 6.0 also hashes in the thread name, so that the same object in differen threads has its own stream. Should I do that?

[12] Note to me: GOTTA implement this!

Section 8.3 Interface

Section 8.3.1 Required Initialization

Before using any random numbers, you must initialize the generator. The path to the master seed file is passed in. This file, if it exists, will be searched for a dictionary entry of master-seed. If found, this will be the master seed.[13]

```
void trandom::init (const std::string & master_seed_path)
                    [static]
```

Section 8.3.2 Common macros

Before describing the trandom class, you should know about some common macros. These are the ones that will be used a lot. The RAND_8 and RAND_32 generate random numbers of the appropriate bit length. The RAND_RANGE() macro generates a bounded random number.

```
RANDOM_RANGE (output_value, min_value, max_value);
RAND_8 (output_value);
RAND_32 (output_value);
```

Section 8.3.3 Creating, copying and destroying a trandom

Once the random number generation is initialized, you can build `trandom` objects. The connstructor takes in a string and a line of which both are optional. However, omitting both will create the identical random number generators. The default copy constructor and `operator=()` can be used to copy a random number stream or set one stream to match another. As there is no dynamic data created, there is no destructor[14].

```
trandom (const std::string & file, uint32 line)
```

Section 8.3.4 Getting random numbers

The draw() method is used to draw a random number. The number will be between 0 and 1 specifically [0..1).

```
double draw ()
```

[13] Note to me: This is really lame. I've gotta implement a direct integer one and not store it in the file!

[14] Note to ne. Should put a virtual one in there. Subclasses might need it.

Section 8.3.5 Creating, copying and destroying a random_range[15]

A simple derived class of trandom is random_range. This class shifts the 0..1 double into a uint32 based range.

```
random_range (const std::string &, uint32)
```

Section 8.4 Examples

1. Super stable random number streams:

As a general technique, it is useful to enclose each call to a random number generator (RAND_RANGE, RAND8, etc) within a static function at the top of a file. This is because the default macros use __FILE__ and __LINE__ as the seed and we don’t want those changing. For example:

```
static uint32 get_next_channel (uint32 a_min, uint32 a_max) {
uint32 r; RAND_RANGE (r, a_min, a_max); return r; }
};
```

2. Direct use of the trandom class

You could also just bypass the macro and provide your own string (or number). This would also be stable across code changes. For example:

```
trandom my_random (“Hello World”, 0);
uint32 my_random_value = my_random.draw ();
```

3. Cycling through random numbers

Sometimes it is necessary to track which random number has been handed out. This is usually when either we want to walk an entire range before returning to a previous value, or we want to make sure we never hand out a duplicate. Another common case is when the number represents a resource, like an endpoint number, which can be “allocated” and “released” randomly during a simulation.

In this case, a boolean array of the appropriate size may be used. Alternately, a map can be used to map the random number to a boolean. For example, assume that we are handing out uint8s. Using the technique in example 1 to isolate random number streams in a static function at the top of the file, we could have:

```
static uint8 next_channel () {
  uint8 x;
  uint32 deadlock = 100000; //This is good enough for most cases.
  static bool used[256] = {0};
  do {  RAND_8 (x);}while (--deadlock &&used[x]);
```

[15] Shouldn’t this be trandom_range ?

```
    if (!deadlock) vout::get (error) << :Deadlock! no more channels at"
                                            << __FILE__ << __LINE__ <<
endl;
  }
};
```

Note that the deadlock ensures that the system never is hung up. This is an important point whenever constraining a random value. As the interactions of the random numbers increases, so does the possibility of a deadlock.

4. Selecting a boolean via a 0..100% probability

Sometimes it's useful to skew a random distribution so that it's not gaussian. There are many ways to do this. For a single bit (or a boolean), one simple way is by providing a threshold, which represents probability of success. Then by generating a random number between 0 and 99, and comparing it to the threshold, we can skew a distribution.

```
static bool get_use_tone_detection () {
  uint8 threshold = dictionary::get ("tone_detection", 50); //default is
50%
  uint8 x; RAND_RANGE (x, 0, 99);
  return (x < threshold);
};
```

5. Selecting a uint32 via a probability distribution

A way to control the distribution of a uint32 is to choose a random number that include all the possible values and then map these values on to a final value[16]. For example:

```
#include <algorithm> using namespace std;
typedef enum valid_address_type {config, io, cached, uncached};
typedef address_probability pair<valid_addresses, uint32>
static valid_address_type get_next_address
(std::list<address_probability> addresses) {
static total_range  =sum<std::list<address_probability>> (addresses)
uint32 x; RAND_RANGE (0, total_range -1);
return find<std::list<address_probability>> (addresses, x).first;
};
```

6. **Subclassing** `trandom`

Yet another way is to subclass the `trandom` or `trandom_range` object and apply a post-processing step. For example, taking the log of the result of the `draw ()` method will produce a logorithmic distibution.

[16] Note to me SOMEBODY CHECK THIS! I am winging it here!

Section 8.4.1 Constrained Random Testing

Often it is useful to constrain the generation of random numbers via an external to the test mechanism. This allows one test to have several different runs. One way to do this is via the master seed picked differently for each run. Another way is to use a test file that provides variables to be used as bounds in the thresholds or distributions shown in the examples above. See the dictionary namespace in Section 10.3.2 for some examples of how to get external variables into your test.

Chapter 9 Accessing Memory

Section 9.1 Overview

Almost every chip that is verified has internal memory or interacts with external memory. This chapter covers Teal's capability for testing those interfaces. It discusses what types of memory is supported and how these are mapped to integer address ranges. It discusses how memory building blocks can be used to support error injection, grouped memory and bank/front door memory access.

Section 9.2 Theory of Operation

For simulation, the storage for memory can be implemented in c/c++ or in the HDL. Implementing the memory in c/c++ is appropriate for extremely large memories, when all accesses must be checked, or when statistics must be gathered. Since a c/c++ implementation could be built on top of Teal (using the vreg class, the run_loop class (see Chapter 12), and the `memory_bank` class (see below), it is not discussed. This chapter is concerned with memory implemented in the HDL. It is assumed that the memory is implemented as a HDL register array.

There are two ways to get access to HDL implemented memory. One is to give Teal the complete path to the memory, as is done for `vreg`[17]. Another is to put a hook task into the module that contains the register bank (see `teal_memory_note()`). In either case, a `memory_bank` object is created for each hook function call. This object contains code to access the internal memory of the simulator that is implementing the HDL register array. The `memory_bank` created by the hook function contains `to_memory()` and `from_memory()` methods, which carry out the access *without advancing simulation time*. The memory_bank also has low and high address 64 bit integers, which are used to allow access via an integer address.

The `memory` namespace keeps track of all memory banks. Note that memory banks are the workhorses of the memory namespace. They do the actual work; the namespace just figures out which one to hand the access request to. In order for this to happen, the memory banks must be mapped into an integer address range. The `memory::map()` function maps a `memory_bank` to an address range. You could put this mapping in the initialization code of your `user_main()`. That way, all other code can just read/write by integer address.

Sometimes several memory banks are grouped together to provide one logical memory bank. In this case, you need to write a special memory bank that first retrives all the sub-

[17] Note to me: BUG: This should be trivial to implement! Do it

memory banks (using `memory::bank_lookup ()`) and then adds itself to the memory namespace's list of memory banks (using `memory::add_bank()`).

Sometimes you want to inject errors in the memory system. This is accomplished in a similar manner to the grouped case, where you looking the bank in question, and add a new bank that handles the memory accesses.[18]

Often it may be beneficial to randomly use either front door access (via a memory bank that uses a bus transactor, like pci or ahb) or back door access via a memory bank that is connected to an HDL model. The front door access is slower and takes simulation time, but front door access will test that path and may also test contention.

Finally, because memory is accessed by integer address, a silicon based memory bank implementation can access the memory with just a pointer de-reference[19].

Section 9.3 Interface

Section 9.3.1 Memory functions

The memory manager has a fairly simple interface. You can map memory to some integer range. You can also add banks of memory for special purpose memory. Later, other parts of your simulation can retrive this memory by address or by path. And yes, you can read and write this memory!

```
void add_map (const std::string & path, uint64 first_address,
                        uint64 last_address)
void add_memory_bank (memory_bank *)
memory_bank * lookup (const std::string & parial_path)
memory_bank * lookup (uint64 address_in_range)
reg read (uint64 global_address)
void write (uint64 global_address, const reg & value)
```

Section 9.3.2 Creating, copying and destroying a memory_bank

A `memory_bank`, at the lowest level, conects to the HDL DUT. This class is internal to the memory namespace and is created whenever a note_memory_bank is placed in the HDL. However, you may want to aggregrate or add special functions for a memory range. In this case, you want to create your own memory bank. The only parameter is the name of the memory bank.

[18] note to me: really want a forget_bank call that takes the bank out if the runing. Can either take it out of the list, or add a boolean to memory_bank like is_subbank and have the lookup skip these types (or do them on a second pass if we init the bank type to subbank?).

[19] Note to me: Build a silion memory map, create one automatically in the silicon memory.cpp.

```
memory_bank::memory_bank (const std::string &)
```

Section 9.3.3 Determining the right memory bank

In order for the memory functions to work, they must determine which memory_bank object will handle the access. This is done by the `contains()` methods. Note that there are two, once for lookup by address and the other for lookup by name.

```
bool memory_bank::contains (uint64 address) const
bool memory::memory_bank::contains (const std::string & path)
const
```

Section 9.3.4 Actually reading and writing memory

Once a bank has been selected, the memory function will call one of the following two methods.

```
virtual reg from_memory (uint64 address) [pure virtual]
virtual void to_memory (uint64 address, const reg & value)
[pure virtual]
```

Section 9.3.5 Examples

1. Simple memory use

```
..assuming, in tb.v... :
initial $teal_memory_note() //in module tb.memory_1
initial $teal_memory_note() //in module tb.memory_2
...in a main.cpp...
memory::add_map ("memory_1", 0x100, 0x200);
memory::add_map ("memory_1", 0x201, 0x400);
memory::write (0x10a, 22); //write local offset 0xa in memory_1 to 22.
if (memory::read (0x10a) != 22) {
  vout::get(error) << "At memory_1[" 0xa << "] " got "
                   << memory::read (0x10a) << " expected 22."
}
memory::write (0x20b, 33); //write local offset 0xb in memory_2 to 33.
if (memory::read (0x20b) != 22) {
  vout::get(error) << "At memory_2[" 0xb << "] " got "
                   << memory::read (0x20b) << " expected 33."
}
```

2. Directly interacting with a memory_bank

```
..assuming, in tb.v... :
```

```
initial $teal_memory_note() //in module tb.memory_2
...in a main.cpp...
memory_bank* bank = memory::lookup (“memory_2”); //partial path
bank->to_memory (0x10, 44);  //directly write a 44 to local offset 0x10
if (bank_from_memory (0x10) != 44) {
  vout::get(error) << “At memory_2[“ 0x10 << “] “ got “
                   << memory::read (0x10) << “ expected 44.”
}
```

Chapter 10 Concurrency In Teal

Section 10.1 Overview

Verification is a complicated task. If a complicated task can be broken down to a set of simpler, independent tasks the problem becomes less complicated. These tasks are then concurrent, and their creation, management, and interaction is the subject of this chapter.

Section 10.2 Theory of Operation

As discussed in Chapter 4, the `user_main`() c function is your top-level controller thread. Generally, this thread waits for the DUT to be ready and then initializes the dictionary and random number subsystems. After that, a series of generators/checkers/transactors are started and some end condition is tested. When the end condition occurs, all threads are stopped and the main thread exits.

The `user_main()` creates threads using the `run_thread`() function. `Run_thread()` takes in the function you want to run and a pointer to a data area. A thread is created and the function is called. Normally, the thread never returns, it is cancelled by your “main” thread with `kill_thread()`[20][21]. If, however, the thread is a temporary one, and thus returns, the last line must be `note_thread_completed()`. Teal needs this call for internal reasons.

When a thread is started, including `user_main()`, it runs until it reaches a waiting point. A waiting point tells Teal that control can be returned to the HDL simulator. Once a wait condition has been satisified, the blocked thread runs. There are three types of waiting points. They are `at()`, `mutex::enter()` and `semaphore::wait()`.

The `at()` function is the most common wait point. It is intended to model the “@” statement in Verilog. It takes in a sensitivity list of `vreg` signals. The signals are matched on the posedge, negedge or any change. A statement such as `at (posedge (clk) || change (reset_n));` would mean to pause the thread until the clk signal went from an X, Z, or 0 to a 1 or any change occurred in the reset_n signal. Execution would then continue after that statement.

The `mutex::enter()` wait point is used when two or more threads need access to some common hardware resource. A good example is the PCI or AHB bus in the system. In the test system and in the software, there may be many threads competing to access the bus(as the same master). The `mutex::enter()` call waits until no other thread is using that

[20] Note to me: should kill_thread be renamed to stop_thread and run_thread be start_thread?

[21] The exception to this rule is generally the user_main() thread, which spawns other threads and then returns.

mutex object and then locks the object. After your thread is done using the hardware, you must `call mutex::unlock ()` to tell Teal that you are done. At that point, any other waiting thread is allowed to access the hardware. See the utility object `mutex_sentry()` for simple mutex management.[22]

The last wait point is `semaphore::wait ().` The semaphore object is intended to loosely model the event object in Verilog. This is most often used for inter-thread communication. For example, suppose you have a monitor thread and your test is waiting for an ACK packet to be sent. You would declare a semaphore object for ack in the monitor object (which also is a separate thread) and have a method `wait_for_ack()` that calls `semaphore::wait()` on the ack semaphore. When the main loop of the monitor object sees an ack on the wire, it would call `semaphore::signal ()` on the ack object. This would cause the `wait_for_ack()` method to return and the calling thread to continue running.

Note that, for running on silicon, the `at()` clause is not supported. This is because `vregs` do not exist in the silicon[23].

The synch module also provides the function `vtime(),` this returns a uint64 of the current simulation time.[24]

The function called `thread_name(pthread_t)` returns the name associated with the thread id, which is returned by `run_thread()`[25].

Section 10.3 C/C++ Interface

Section 10.3.1 Creating and Destroying Threads

A new task is created with `run_thread()`. This function takes in a function to run, a data parameter and a task name. It returns an id to be used to kill the thread (or any other pthread function). The function `kill_thread()` takes in a thread id and stops that thread.

In general, a thread runs until it is stopped by another thread. If however a thread runs to completion and returns from the user_thread function, it must call `note_task_completed()` to let Teal clean up internal data structures[26].

---

[22] Note to me: Code THIS! Test this in mutex_test

[23] Note to me: create a CSR utility class that takes in a vreg and an address.

[24] Note to me: add a time_in() function that takes in an enum and converts all vtime measurements to that timebase. Does this mean vtime should be a double? If so, operator== gets weird. If no, we could loose precision. At any rate, shouldn't we move this into the utilities area?

[25] Note to me: Well here is an intersting point about a teal.h general header. People don't care that this is a synh thing. It's really a utility thing!

The thread_name() function converts between the thread_id and the task name.

```
void (user_thread) (void* user_data)
pthread_t run_thread (user_thread, void * user_data, const
std::string & name)
void * kill_thread (pthread_t)
void note_task_completed ()
std::string thread_name (pthread_t)
```

The `teal_main_misc()` function is the actual verilog hook to convert the $teal_main call in the HDL testbench to this c routine. It must be in a pli or vpi call back structure. See the figure below.

```
void teal_main_misc (int user_data, int reason)
```

The `vtime()` function returns the current simulation time.

```
uint64 vtime ()
```

The `at()` function is the main way a task pauses. It is given a sensitivity list, which is an object that is automatically created (and destroyed) by calling `posedge()`, `negedge()` or `change()` with a vreg as its argument. The list is also extended when you have multiple `vreg` changes (`negedge`, `posedge`, or `change`) seperated by the `||` operator.

NOTE: `posedge()` and `negedge()` only test the lowest bit for the appropriate edge. If you need to test for a different bit, you must build a vreg on a specific bit.[27]

```
void at (const sensitivity &)
posedge (vreg & v)
negedge (vreg & v)
change (vreg & v)
```

Section 10.3.2 Creating and Destroying a Mutex

Often several independent threads need to use a common hardware resource, such as a bus. In the case that each thread has it's own master id and the bus has hardware arbitrartion, each task can arbitrate for the bus. However, if the independent tasks are the same master (as if they are emulating multiple software threads on a cpu), they need a simulation mechanism to arbitrate. This is the purpose of the mutex class. Once constructed, it's pointer is passed to all affected modules (or hidden in a common transactor, like bus master). Each task calls `lock()` to either gain access to the hardware or block until the

[26] Note to me: Gotta write and test a kill_all_threads()

[27] Note: It would be a simple matter to have a second constructor on posedge/negedge that take in a bit position and watch that bit.

current task is finished with the hardware. Once the `lock()` method returns, you access the hardware and then call `release()` to inform Teal that you are done with the hardware. At that point, any waiting threads re-arbitrate for the mutex.

The constructor only takes in a name for the mutex.

```
mutex (const std::string & name)
~mutex ()
```

Section 10.3.3 Working with a Mutex

There are only two operations on Mutex. One is to acquire the mutex, by calling lock() and the other is to release the mutex by calling unlock().

```
void lock ()
void unlock ()
```

Section 10.3.4 Creating and destroying a Condition

Whenever you have multiple tasks, there is a good chance that they will need to communicate. While tasks can put data into work queues and take data out, there must be some mechanism to signal that one task as just put some data in (or taken something out). Also, a monitor task may need to announce a particular condition (like ack, nak, or byte_sent). There may or may not be a receiver to note the announced event. The `condition` class provides such inter-task communication[28].

A `condition` isgiven a name when constructed.

```
condition (const std::string & name)
~condition ()
```

Section 10.3.5 Working with a condition

Once a condition is created, it's pointer is passed to the affected tasks (or buried in a method of a class, like `monitor::wait_for_ack ()`). At the appropriate time, one task calls `wait()` and it blocks until another thread calls signal.

```
void signal ()
void wait ()
```

[28] Note to me: BUG have to remove the got to wait after signalled logic or modify it to wait at least one simulation step by noting the vtime() and doing a return only of vtime is > signal time. HAVE TO THINK ABOUT THIS.

Section 10.3.6 Examples

1. Simple at() expressions

```
vreg address (“testbench.address”);
vreg clk (“tb.clk”);
at (posedge (clk) || change (address)); //wait until next rising clk or
any address change
at (negedge (clk)); //falling edge test
```

2. Mutex

```
mutex main_bus_mutex (“main bus”);
run_thread (master_one, &bus_mutex, “task one”);
run_thread (master_two, &bus_mutex, “task two”);
void master_one (void* context) {
  mutex* m = static_cast<mutex*> context;
  m.lock ();
  vreg address (“tb.address”); address = 0x10;
  vreg clk (“tb.clk”);
  at (posedge (clk)); //wait one clk pulse
  vreg data (“tb.data”); vout << thread_name() << “data is “ << data <<
endl;
}
void master_two (void* context) {
  mutex* m = static_cast<mutex*> context;
  m.lock ();
  vreg address (“tb.address”); address = 0x16;
  vreg clk (“tb.clk”);
  at (posedge (clk)); //wait one clk pulse
  vreg data (“tb.data”); vout << thread_name() << “data is “ << data <<
endl;
}
```

3. Semaphore

```
struct context {
  void context (): mailbox (“main mailbox”),{};
  semaphore mailbox;
  std::vector <uint32> work_queue;
}
context my_context;
run_thread (producer, context, “producer”);
run_thread (consumer, context, “consumer”);
```

```
void producer (void* c) {
  context* the_context = static_cast<context*> ( c );
  the_context.work_queue.push_back (10);
  the_context.mailbox.signal ();
  the_context.work_queue.push_back (20);
  the_context.mailbox.signal ();
}
void consumer (void* c) {
  context* the_context = static_cast<context*> ( c );
  for (;;) {
    the_context.mailbox.wait ();
    vout::get (note) << thread_name () << received <<
the_context.work_queue.front ();
    the_context.work_queue.pop_front ();
  }
}
```

Chapter 11 Dictionary Namespace

Section 11.1 Overview

This chapter introduces a namespace to solve a common problem in simulation: getting test parameters.[29]

Section 11.2 Theory of Operation

Sometimes you want to control a test via an external file. This allows a single test to have many different directed runs. These runs would still use random numbers, except that their range might be constrained by the test file. For example, the probability of an feature, or error may be an external variable. Alternatively, maybe your test file can be used to turn on and off features.

Teal provides a simple dictionary namespace for this purpose. After the dictionary namespace is initialized with a filename, the file is opened and the first word in every line is cached. Then, your test can query the dictionary (with the `find`() function) and see what the value after the keyword is. For example, if the file had "number_of_streams 33" on a line, a call to `dictionary::find ("number_of_streams");` would return 33.

Section 11.3 C/C++ Interface

Section 11.3.1 Dictionary main functions

In order to use the dictionary, you must first call start(). This function opens the file and indexes all the entries. When you are done with the file, call stop().

```
void start (const std::string & path)
void stop ()
```

Section 11.4 Working with the dictionary

In order to see if an entry exists, call the find() function that returns a `std::ifstream`. If the stream is bad (using the `std::bad()` function in `iostreams`) the entry does not exist. If you are looking up an integer and just want a default value to be used if there is no entry, call the find that returns a uint32.

```
uint32 find (const std::string & name, uint32 default_val)
std::ifstream& find (const std::string & name)
```

[29] Note to self: Gotta add an include directive so that dictionary can be a hierachy, Also, consider merging in command line parameters and dictionary. This would imply virtual functions and a define new() somewhere.

11.4.1.1 Examples

1. Simple at() expressions

```
dictionary::start ("teal_test.txt"); //assume it has a line "foo 10"
vout::get(note) << "Foo is " << dictioary::find ("foo", 20) << endl;
std::ifstream bar (dictionary::find ("bar));
if (bar.bad ()) {
  vout::get(warning) <, "bar not found" << endl;
}
else {
  double bar_val; bar_val << bar; //read it in, use bar_val<< std::hex <<
bar; if the valus is in hexidecimal
  vout::get(note) << "Bar is " << bar_val << endl;
}
```

2. A min max integer

```
mutex main_bus_mutex ("main bus");
std::ifstream bar (dictionary::find ("channel_range));
if (bar.bad ()) {
  vout::get(warning) <, "channel range not found" << endl;
}
else {
  uint32 min_val (0);
  uint32 max_val (0);
 min_val << max_val << bar;
  vout::get(note) << "Min is " << min_val << " and max is " << max_val <<
endl;
}
```

Chapter 12 The run_loop object

Section 12.1 Theory of Operation

While the run_thread() and kill_thread() functions are adequate, it is extremely common to have an object that represents a checker, generator, or transactor[30]. All of these types of objects have a common theme. You create them, you start them and then sometime later, you stop them. Once they are started, they are in an infinite loop that has a wait point statement and then a loop body[31]. The run_loop class provides exactly this functionality.

The run-loop constructor takes in a path to the HDL root for this object and a name for the created thread. It has virtual `start()` and `stop()` methods, which, by default, start and stop a thread. That thread is an infinite loop that first calls the pure virtual method `do_at ()`, and then calls `one_iteration ()`.

Section 12.2 Interface

Section 12.2.1 Constructing, copying, and destroying a run_loop

The `run_loop` construction takes in a path to be used to hook up to the HDL and a name to be used for the task that is created. The copy constructor is priovate to prevent copying a `run_loop` object[32].

```
run_loop (const std::string & path, const std::string name)
virtual ~run_loop ()
```

Section 12.2.2 Working with a run_loop object

Since a `run_loop` abstracts a common infinate loop, calling `start()` creates and runs the independent task running. The `stop()` method is used to stop the task.

It is up to the implementor of a derived class to determine what to put in the `do_at()` and the `one_iteration()` methods. Maybe you are implementing a monitor and the `do_at()` is just a `at(negedge (reset_n) or posedge (clk))` and the `one_iteration()` is a state machine of the protocol. Or maybe this is a consumer similar to the semaphore example and the `do_at()` is just a `semaphore.wait()` and the `one_iteration()` pops a value from a work queue and processes the data.

```
virtual void start ()
```

[30] Note to self, the generator may not need an hdl path, the checker may or may not depending on the level.

[31] Note to me: Should the method name be loop_body() ?

```
virtual void stop ()
virtual voiddo_at () = 0;
virtual void one_iteration = 0;
```

Section 12.3 Examples

Simple one as in test, and an interrupt handler with different subclasses for simulation and silicon.

[32] Note to me; Gotta implement this!

Chapter 13 Appendix

Section 13.1 Teal Coding Conventions

Teal follows a few simple rules in the interface (and implementation). All code components are within the teal namespace, with the exception of the user_main() function, which is implemented by you.

Within a class, public methods and data members come first, followed by protected and then private members. This rule may be broken is a public member is closely tied to a protected member.

In general, all compiler-generated methods are defined in an appropriate access area. For example, in the vout class, one is not supposed to copy or assign a vout, so those methods are both not implemented and placed in the private scope.

Some objects only have one instance by design. If so, there is a static method, called get() that returns a reference to the singleton.

Any data or method that is intended to be in the protected or private scope has a trailing underscore.

Inline methods are generally not used, but may be present if it seems more elegant. Obviously, that is a judgement call.

Very few MACROS are used. They exist mainly where the macro expansion, as when using __FILE__ (see trandom.h)

Section 13.2 Building Teal from Source Code

In general, set the SIM and VERILOG_INST_DIR and type make. The define vpi_2_0 causes Teal to use the vpi interface, omitting it uses the 1.0 interface. The output is an archive called libteal_$(SIM).a