What Time Is It: Implementing a SystemVerilog Object-Oriented Wrapper for Interacting with the C Library `time`

Eldon Nelson M.S. P.E.
Synopsys, Inc. (eldon_nelson@ieee.org)

***Abstract*–Simulation time is the time within the SystemVerilog simulation which can be obtained by running the built-in `$time` function. Wall-time is the time you see on your watch; the actual time in space that we are all experiencing together. SystemVerilog users are attempting to mark when a test started in the simulation log for post processing, determine how long in wall-time a sequence takes to run, or conduct simulator performance experiments. The wall-time query with most other programming languages is common and easily answered. Unfortunately, SystemVerilog [1] does not have access to wall-time, out-of-the-box. This lack of the wall-time query has resulted in less than optimal understanding and solutions to get to the wall-time from the SystemVerilog simulation. This paper discusses design choices and motivations, and provides the source code for a user-friendly SystemVerilog object-oriented wrapper to interact with the C library `time` [2]. The proposed approach is several hundred times faster than conventional `$system` functions. In addition, an approach to determining key bottlenecks in simulation runs is proposed by plotting the simulation time versus wall-time.**

I. Introduction

The top hit in search engines for "wall time systemverilog" yields this Stack Overflow [3] article [4], where the top-voted answer is incorrect. The Stack Overflow article proposes directly using `$system("date").` The result is the return code of the shell function - and not the actual wall-time! The return code is the number zero, which in Linux means the operation completed without errors. This is not the desired data. The second solution proposed by the same Stack Overflow article achieves the answer, but in an inefficient way. The second solution writes to the file system the text returned by the shell command `"date"`, opens a file handle, and finally reads the contents of the file to a variable. While using the SystemVerilog `$system` function is a viable solution, it is not ideal.

The problem is that there is not a clean way to get the wall-time in SystemVerilog. In most other programming languages getting the wall-time is a built-in feature of the language or available in a first-class library. The proposed solution is to develop a SystemVerilog object-oriented wrapper of the C library `time` which can be accessed via the SystemVerilog DPI (direct programming interface).

FIGURE 1 QUESTION AND FIRST ANSWER FROM STACK OVERFLOW ON GETTING LOCALTIME [4]

There are many design choices that will go into making this proposed solution friendly for users. The first choice is to make it an object-oriented wrapper. This means we will not only be making raw DPI call into the C library `time` functions – although for many that may be sufficient. We will instead devise a strategy that will make interacting with the C `time` object-oriented and conform to the expectations of modern programming techniques. Fortunately, there are programming languages which already have acceptable and polished object-oriented wrappings for working with general wall-time queries. We will base our solution upon the choices of the Python `time` library [5] and the Ruby `Time` library [6].

In this paper, the solution is built-up gradually with many working stops in between. Hence, making it possible to stop at earlier levels of abstraction. The paper ends by implementing a user-friendly, object-oriented wrapper of the C library `time`. The direction chosen is more ambitious than wrapping a single `time` function. Emulating the methods of working with wall-time, as implemented in other programming languages, provides a richer solution that can do more for the user. The motivations of why would this approach was chosen is discussed. The cross-platform solution from this paper `svtime` [7] is available under the GNU General Public License V3 (GPL) [8] and is distributed on GitHub [9]. You can access the svtime solution from the following location:

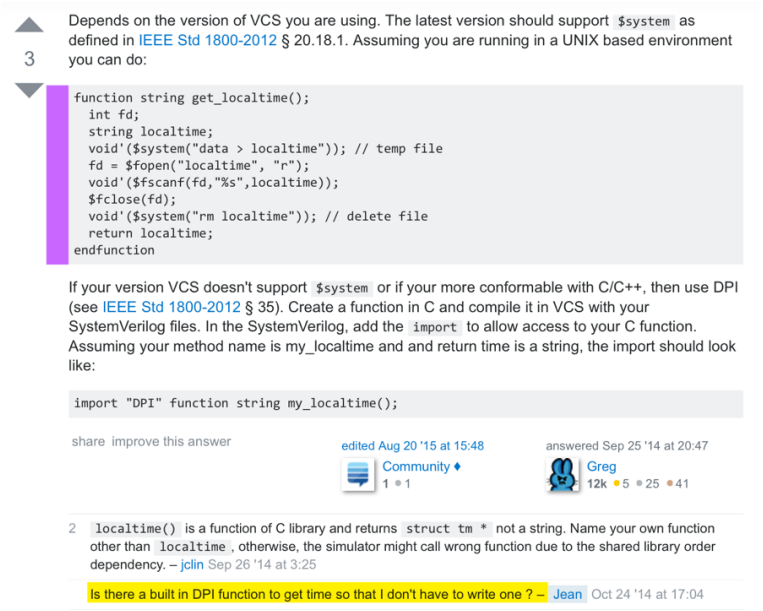
`https://github.com/tenthousandfailures/svtime`

The need to get the wall-time during a simulation is universal. You should be able to write to the simulation log the wall-time when certain events have occurred or to be able to have timers that are based on wall-time. The implemented solution can be used by others to answer that need. The paper also shares the performance of this object-oriented solution to existing solutions of getting wall-time. An application of this library is demonstrated that plots simulation time versus wall-time as an aid for simulator performance profiling.

II. Approach

When dealing with wall-clock time on computers there is a concept known as the epoch. The epoch is defined as 1970 on Linux systems. When dealing with time in many different programming languages, the seconds-since-the-epoch is used as a method to measure wall-time. In addition, from this abstraction of using the seconds-since-the-epoch, many different convenient time manipulations are achieved.

First, let us look at the functional, yet inefficient second-place answer on Stack Overflow. This answer is closer to what we want to achieve.



3

Depends on the version of VCS you are using. The latest version should support `$system` as defined in IEEE Std 1800-2012 § 20.18.1. Assuming you are running in a UNIX based environment you can do:

```
function string get_localtime();
  int fd;
  string localtime;
  void'($system("data > localtime")); // temp file
  fd = $fopen("localtime", "r");
  void'($fscanf(fd,"%s",localtime));
  $fclose(fd);
  void'($system("rm localtime")); // delete file
  return localtime;
endfunction
```

If your version VCS doesn't support `$system` or if your more conformable with C/C++, then use DPI (see IEEE Std 1800-2012 § 35). Create a function in C and compile it in VCS with your SystemVerilog files. In the SystemVerilog, add the `import` to allow access to your C function. Assuming your method name is my_localtime and and return time is a string, the import should look like:

```
import "DPI" function string my_localtime();
```

share improve this answer

edited Aug 20 '15 at 15:48 Community ♦ 1 • 1

answered Sep 25 '14 at 20:47 Greg 12k • 5 • 25 • 41

2 `localtime()` is a function of C library and returns `struct tm *` not a string. Name your own function other than `localtime`, otherwise, the simulator might call wrong function due to the shared library order dependency. – jclin Sep 26 '14 at 3:25

Is there a built in DPI function to get time so that I don't have to write one ? – Jean Oct 24 '14 at 17:04

Figure 2 Second-Place Answer Stack Overflow for Getting Time [4]

In Figure 2, we see the proposed solution (in purple highlight) runs the Linux command `date` (corrected for mis-spelling on Stack Overflow), then opens the resultant output file, and finally reads the contents of the resultant output file into a string called `localtime`. The problem with this approach is that it requires several inefficient and possibly contradictory steps. Using a temp file on the file system, which could potentially be accessed from many threads, opens the possibility for corruption and locking of that temp file. The performance of opening, reading, and removing the temp file is orders of magnitude slower than our final solution. This paper will answer the question (highlighted in yellow above):

> *"Is there a built in DPI function to get time so that I don't have to write one?"*
> *User Jean on Stack Overflow, Oct 14, 2014* [4]

The first step towards the solution is to wrap the C library `time`. This library has a very simple data structure that is used in many of its functions, which is shown below.

```
typedef struct {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
}  struct_time;
```

FIGURE 3 TIME DATA STRUCT

This data structure is a collection of 32-bit `int(s)` that represent time since the epoch. For example, if all values in this `struct` were `0`, the time would be `January 1, 1970 00:00:00 am`. The use of the C `time` library is thoroughly documented. The C wrappers this paper created to provide SystemVerilog compatible interaction is provided in the source code of `c_func.c` [10]. To understand how this works, this paper demonstrates using the user API to interact with the C `time` library from SystemVerilog.

```
struct_time s1;
...
s1 = svtime::localtime();
$display("\t svtime::asctime(s1): %s", svtime::asctime(s1));
$display("\t s1.tm_sec = %0d", s1.tm_sec);
$display("\t s1.tm_min = %0d", s1.tm_min);

svtime::sleep(2);
s1 = svtime::localtime();
$display("\t svtime::asctime(s1): %s", svtime::asctime(s1));
```

FIGURE 4 C/PYTHON STYLE STATIC TIME USAGE OF SVTIME [11]

```
        svtime::asctime(s1): Thu Nov 16 01:37:09 2017
        s1.tm_sec = 9
        s1.tm_min = 37
        svtime::asctime(s1): Thu Nov 16 01:37:11 2017
```

FIGURE 5 OUTPUT OF C/PYTHON STYLE STATIC TIME USAGE OF SVTIME

We can see that the C `time` functions, `localtime` and `asctime`, are ported to return: a time data `struct s1` (in Figure 4) and a formatted string respectively. This behavior matches the Python usage that uses the same names. Many other functions, such as `strftime`, `mktime`, and `sleep`, are ported in the same way to match the C and Python usage. This is the goal! There is no need to reinvent these API calls because they were defined decades ago and already exist within the C `time` library.

Now, let us take this solution to the next level. There is not anything object-oriented about the way the above commands are specified. In fact, these are all static functions that take an input and return an output. We can look at the Ruby `Time` library, which expands upon this approach, but provides an object-oriented approach to working with time.

```
svtimep svtimep_inst;
...
svtimep_inst = new();
svtimep_inst.now();
$display("\t svtimep_inst.to_s() = %s", svtimep_inst.to_s());
$display("\t svtimep_inst.sec() = %0d", svtimep_inst.sec());
$display("\t svtimep_inst.min() = %0d", svtimep_inst.min());

svtime::sleep(2);
svtimep_inst.now();
$display("\t svtimep_inst.to_s() = %s", svtimep_inst.to_s());
```

FIGURE 6 RUBY STYLE OBJECT-ORIENTED TIME USAGE OF SVTIME [11]

```
        svtimep_inst.to_s() = 2017-11-16 01:37:07
        svtimep_inst.sec() =           7
        svtimep_inst.min() =          37
        svtimep_inst.to_s() = 2017-11-16 01:37:09
```

FIGURE 7 OUTPUT OF RUBY STYLE OBJECT-ORIENTED TIME USAGE OF SVTIME

Figure 7 has the same content as Figure 5. The default formatting of the two differs, but the wall-time content is the same. The usage styles show the difference between a non-object-oriented approach to an object-oriented approach to wrapping the C `time` library. Both are valid, and depending on style preference, you can use either approach. It is demonstrated later in the paper that these two approaches are very similar as far as simulator performance; being within `3%` of each other. My opinion is that Figure 6, the object-oriented approach, is more user-friendly and makes available easier conversion and automation. Both methods are provided in the package. In fact, both interfaces match what the Ruby `Time` library does as well; there, the non-object-oriented version underpins the workings of the object-oriented implementation.

III. Results and Implementation

The `svtime` library provides two implementations, the non-object-oriented and the object-oriented approach. The preferred object-oriented approach is reproduced in excerpt in Figure 8. Figure 8 implements the API functions called in Figure 6.

```
class svtimep extends svtime;
   struct_time tm;

   function new(integer year = 'x,
                integer mon = 1,
                integer mday = 1,
                integer hour = 0,
                integer min = 0,
                integer sec = 0);
      if ($isunknown(year)) begin
         tm = localtime();
      end else begin
         tm.tm_year = year;
         tm.tm_mon = mon;
         tm.tm_mday = mday;
         tm.tm_hour = hour;
         tm.tm_min = min;
         tm.tm_sec = sec;
      end
   endfunction // new

   function void now(longint t = 0);          // grab the latest time or time from user
      if (t == 0) begin
         tm = localtime();
      end else begin
         tm = super.localtime(t);
      end
   endfunction

   function int sec();                        // return the object seconds
      return(tm.tm_sec);
   endfunction

   function int min();                        // return the object minutes
      return(tm.tm_min);
   endfunction

...

   function longint to_i();                   // convert time to seconds since epoc
      return super.mktime(tm);
   endfunction

   function string to_s(string fmt = "%Y-%m-%d %H:%M:%S");
      return(strftime(fmt));
   endfunction


endclass
```

Figure 8 Excerpt of the svtimep Object-Oriented SystemVerilog Implementation [7]

We can see that the `svtimep` class internally keeps track of a `struct` called `tm`. All manipulations of this `struct` are wrapped within the function calls. Internally, these calls eventually resolve to interactions with the C `time` library and the necessary SystemVerilog DPI calls. The user does not need to know this, but can merely look at the class implementation above that mirrors the same API that the Ruby `Time` library provides. For example, if you wanted to know how many seconds there were since the epoch, you would call the method `to_i()` that returns a `longint` with the

value based on what was currently stored in the `tm` struct. For further detail, visit the source code provided [7] or consult the examples provided in the included `Makefile`.

```
> make help

clean                          Cleans up work area
help                           Help Text
perf_non_oo                    Do the SystemVerilog static method performance benchmark
perf_oo                        Do the SystemVerilog object-oriented preferred performance benchmark
perf_shell                     Do the basic Linux shell version of the performance benchmark
perf_stack                     Do the SystemVerilog date wrapper (Stack Overflow) performance benchmark
shared_c                       Create the shared c library c_func.so
vcs_example_build              Build the example in VCS
vcs_example_sim                Simulate the example in VCS

  Examples
    > make clean shared_c vcs_example_build vcs_example_sim
    > make perf_stack
    > make perf_oo
```

FIGURE 9 MAKEFILE USAGE FOR SVTIME [12]

One of the goals of this paper is to show a better alternative than using `$system` functions to get the wall-time. An impressive result of this approach is quantifying how much better it is to use a SystemVerilog DPI call to get wall-time than trying to go outside of the simulator with the `$system` task. Figure 10 shows different methods of getting wall-time. There is an 801 times speed improvement to using the SystemVerilog DPI versus using `$system`. You may need to acquire the wall-time on multiple occasions during the simulation and you may see a tremendous performance benefit from using the proposed method over using the `$system` method. Regardless, the speed difference is improved nearly three orders of magnitude when compared to other approaches.

Method (10,000,000 calls)	Time in Seconds	make command
SV `date` command line	28400	`make perf_stack`
Shell Script `date`	8100	`make perf_shell`
SV object oriented method (Figure 6)	32	`make perf_oo`
SV non-object oriented static method (Figure 4)	31	`make perf_non_oo`

FIGURE 10 COMPARISON OF METHODS FOR GETTING WALL-TIME IN SYSTEMVERILOG

An application of this `svtime` Library is to create the graph, as shown in Figure 11, which plots simulation time versus normalized wall-time per micro-second simulation time. The performance of the simulation in terms of simulation time per wall-time seconds is not a constant. At different points during the simulation the simulator is processing simulation time at different wall-time speeds. This provides an insight into what makes a simulation fast or slow and supplements the simulation profiler data in a different way. This method exposes periods of simulation time that took abnormally long in wall-time, which can then direct performance analysis efforts.

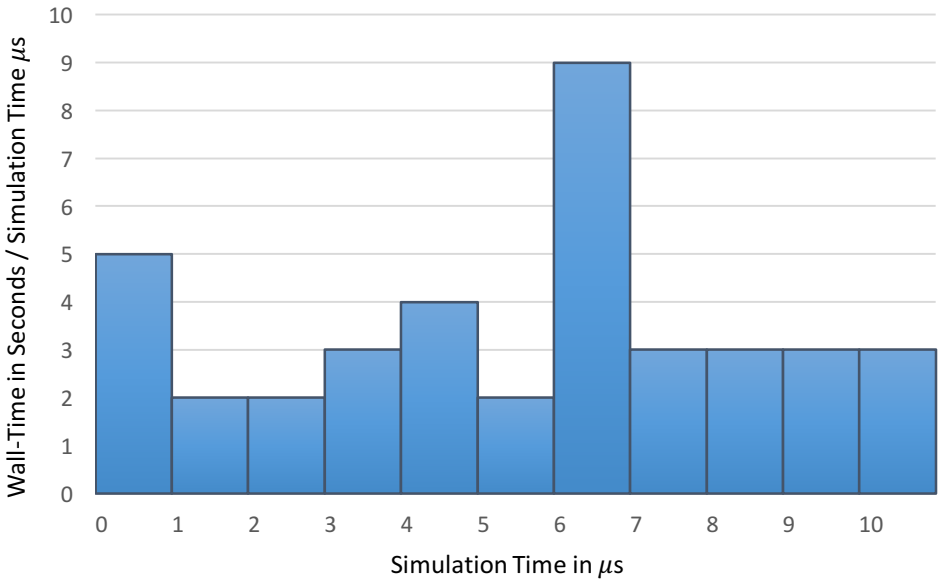


FIGURE 11 SIMULATION TIME VERSUS WALL-TIME

Also, provided is a simple SystemVerilog `interface` named `time_monitor` in Figure 12 that has a single input and a parameterized number of positive clock edges for it to count. When the number of positive clock edges is met, it prints out a `$display` message that states the seconds since the epoc that occurred in addition to the simulation time. You can then process the output into the graph Figure 11.

```
interface time_monitor(input reg clk);
   parameter period = 2;
   parameter prefix = "tm";

   longint   cycles = 0;
   svtime_pkg::svtimep svtimep_inst;

   // initialize the svtimep class
   initial begin
      svtimep_inst = new();
      svtimep_inst.now();
      $display("[%s] time_monitor at simtime %0t initial with period %0d at epoc sec: %0d",
               prefix, $time(), period, svtimep_inst.to_i());
   end

   always @(posedge clk) begin;
      if (cycles >= period) begin
         svtimep_inst.now();
         $display("[%s] time_monitor at simtime %0t with period %0d at epoc sec: %0d",
                  prefix, $time(), period, svtimep_inst.to_i());
         cycles = 0;
      end else begin
         cycles++;
      end
   end
endinterface
```

FIGURE 12 TIME_MONITOR IMPLEMENTATION EXAMPLE [7]

Attaching this `interface` to a testbench to do simulation performance metrics is demonstrated in Figure 13.

```
module top;
...

   reg            clk = 0;

   time_monitor #(10000) time_monitor_inst(.clk(clk));
...
```

FIGURE 13 INSTANTIATING TIME_MONITOR EXAMPLE [11]

Another application is for nightly verification regressions runs to consider the wall-time. For example, it may be a policy that at 6 a.m. every day, all nightly verification regression jobs are killed automatically. To avoid loss of the in-flight verification regression run coverage data, which would be lost if the jobs were manually killed, you can configure the simulation to periodically check the current wall-time. If the wall-time is within a certain range, the simulation can stop the creation of new stimulus and start the procedure to end the test early. To implement such a configuration, use a time query to check if the current wall-time was between 5 a.m. and 6 a.m. as implemented in Figure 14. Then, perform the necessary end of test procedures.

```
interface time_alarmclock(input reg clk);
   parameter period = 2;
   parameter alarm_hour = 5;
   parameter alarm_min  = 0;
   parameter prefix = "ta";

   longint   cycles = 0;
   bit       triggered = 0;
   svtime_pkg::svtimep svtimep_inst;

...

   // trigger if walltime equals alarm_hour and at or above alarm_min
   always @(posedge clk) begin;
      svtimep_inst.now();
      if ((cycles >= period) &&
          (triggered == 0) &&
          (alarm_hour == svtimep_inst.hour()) &&
          (alarm_min <= svtimep_inst.min())
         ) begin
             $display("[%s] time_alarmclock TRIGGERED at simtime \
                      %0t with period %0d at %2d:%2d:%2d",
                      prefix,
                      $time(),
                      period,
                      svtimep_inst.hour(), svtimep_inst.min(), svtimep_inst.sec()
                     );
             triggered = 1;
             cycles = 0;
          end else begin
             cycles++;
          end
   end

endinterface
```

FIGURE 14 TIME_ALARMCLOCK IMPLEMENTATION EXAMPLE [7]

IV. Summary

A common question within the SystemVerilog community is how to get the wall-time during the simulation. There is no built-in method within SystemVerilog to get wall-time. This paper documents the development and motivations of a SystemVerilog object-oriented wrapper of the C library `time`. The design of this wrapper is based upon the object-oriented solutions from the Python `time` library and the Ruby `Time` library. The solution was built up over small steps with working solutions in between. The solution is released under the GNU GPL license and available on GitHub.

The `svtime` package provides a non-object-oriented implementation of the wrapper, which is very similar to the Python `time` implementation. This Python style time library uses purely static methods and a `struct` matching that of the C `time` library. Also provided in the `svtime` package is a Ruby style object-oriented implementation. This uses, in contrast, an object to handle the conversions and functions in an object-oriented style. The author prefers the Ruby style implementation and recommends that version for ease-of-use at a small, but reasonable, overhead of `3%` over the Python style. The performance benefit of using a SystemVerilog DPI wrapper is 801 times faster than a common solution using `$system`, as documented on Stack Overflow [4].

An example use case showing a graph of simulation time versus wall-time, which is easily made possible by this solution, is presented. The insight gained from creating such a graph could help to supplement the simulation profiler data and help pinpoint hangs or periods in the simulation that slowed down the whole simulation.

V. Future Work

The `svtime` library implements about half of the API functions documented in the original C `time`, Python `time` and Ruby `Time` library functions. The current implementation skips some of the functions that did not have high impact for SystemVerilog simulations. One skipped feature that could be useful is to include sub-second timing. The Ruby `Time` library included the ability to get this sub-second timing information, which could help with the resolution that is currently in seconds. From the SystemVerilog code, being able to measure how long certain sub-second wall-time actions took with a low performance overhead would be very useful.

This `svtime` library is intended to be cross platform and should work with any simulator. Proving this compatibility and providing example cases for various simulators would help speed adoption of this library.

VI. Acknowledgment

I thank my employer Synopsys, Inc. for sponsoring my attendance at DVCon to share this paper. I thank Synopsys technical writer Rupen Sharma for an excellent editorial review of this paper. I was inspired by colleges from several companies who asked the question of wall-time in a SystemVerilog simulation. However, it was not until I saw the Stack Overflow answer [4] that I realized this was a major misconception. This motivated me to determine an effective and accurate method for solving the wall-time issue.

VII. REFERENCES

[1] IEEE Computer Society and the IEEE Standards Association Comporate Advisory Group, IEEE Standard for SystemVerilog 1800-2012, New York, NY: IEEE, 2013.

[2] Internation Organization for Standardization and Internal Electrotechnical Commission, "ISO/IEC 9899:TC2," 06 05 2005. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. [Accessed 17 08 2017].

[3] Stack Overflow, "Stack Overflow," [Online]. Available: https://stackoverflow.com. [Accessed 17 08 2017].

[4] Stack Overflow, "Get system time in VCS," Stack Overflow, 25 9 2014. [Online]. Available: https://stackoverflow.com/questions/26046469/get-system-time-in-vcs. [Accessed 17 8 2017].

[5] Python Software Foundation, "Time access and conversions," [Online]. Available: https://docs.python.org/3/library/time.html. [Accessed 17 08 2017].

[6] Ruby community, "Time," [Online]. Available: https://ruby-doc.org/core-2.3.0/Time.html. [Accessed 17 08 2017].

[7] E. Nelson, "svtime Library," 17 08 2017. [Online]. Available: https://github.com/tenthousandfailures/svtime. [Accessed 17 08 2017].

[8] Free Software Foundation, "GNU General Public License," 29 6 2007. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html. [Accessed 26 11 2017].

[9] GitHub, Inc., "GitHub," 2017. [Online]. Available: https://github.com.

[10] E. Nelson, "c_func.c," 2017. [Online]. Available: https://github.com/tenthousandfailures/svtime/blob/master/pkg/c_func.c.

[11] E. Nelson, "top.sv," 2017. [Online]. Available: https://github.com/tenthousandfailures/svtime/blob/master/example/top.sv.

[12] E. Nelson, "Makefile," 2017. [Online]. Available: https://github.com/tenthousandfailures/svtime/blob/master/Makefile.

[13] Synopsys, "VCS Version M-2017.03," 2017. [Online]. Available: http://www.synopsys.com.