

# OVM & UVM Techniques for Terminating Tests

Clifford E. Cummings  
Sunburst Design, Inc.  
www.sunburst-design.com  
cliffc@sunburst-design.com

Tom Fitzpatrick  
Mentor Graphics Corp  
www.mentor.com  
tom\_fitzpatrick@mentor.com

## ABSTRACT

The Open Verification Methodology (OVM) and the new Universal Verification methodology (UVM) have a number of methods for terminating the `run()` phase at the completion of a test, usually via a combination of sequence completion, calls to the global `stop_request` mechanism and/or the recently-added objection mechanism. Many users also use built-in event and barrier constructs on a more application-specific basis to achieve their goals. This plethora of choices has led to some confusion among the user community about how best to manage this important aspect of the testbench.

This paper describes various techniques for gracefully terminating an OVM/UVM test, and proposes a set of guidelines to avoid further confusion.

## Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

## General Terms

Algorithms, Documentation, Performance, Design, Experimentation, Standardization, Languages, Theory, Verification.

## Keywords

UVM, OVM, SystemVerilog, testbench, `global_stop_request()`, `raise objection`, `drop objection`.

## 1. INTRODUCTION

UVM is a verification class library based largely on the OVM version 2.1.1 class library. The descriptions in this paper reference the UVM code and methods but the comments on this topic are just as applicable to OVM. Terminating the simulation `run()` phase is identical using either class library except where noted.

Unlike all of the design modules and interfaces that are called during compilation and elaboration, none of the UVM testbench environment is setup until after simulation starts.

To help understand these topics, it is often useful to understand some basics about how the UVM class library is laid out in the `uvm` directory and file setup. This paper details important basics on vital files and how they are laid out. The location of important files, class definitions and global variables and tasks can be difficult to find. This paper will help to partially navigate the UVM maze that hides many important details.

### 1.1 Version

This paper is based on UVM version 1.0ea (ea - Early Adopter version), which is largely based on OVM version 2.1.1.

## 2. COMPILING DESIGNS & RUNNING UVM

To help understand how UVM simulations work within the SystemVerilog testbench environment, it is useful to have a big-picture view of the entire simulation flow.

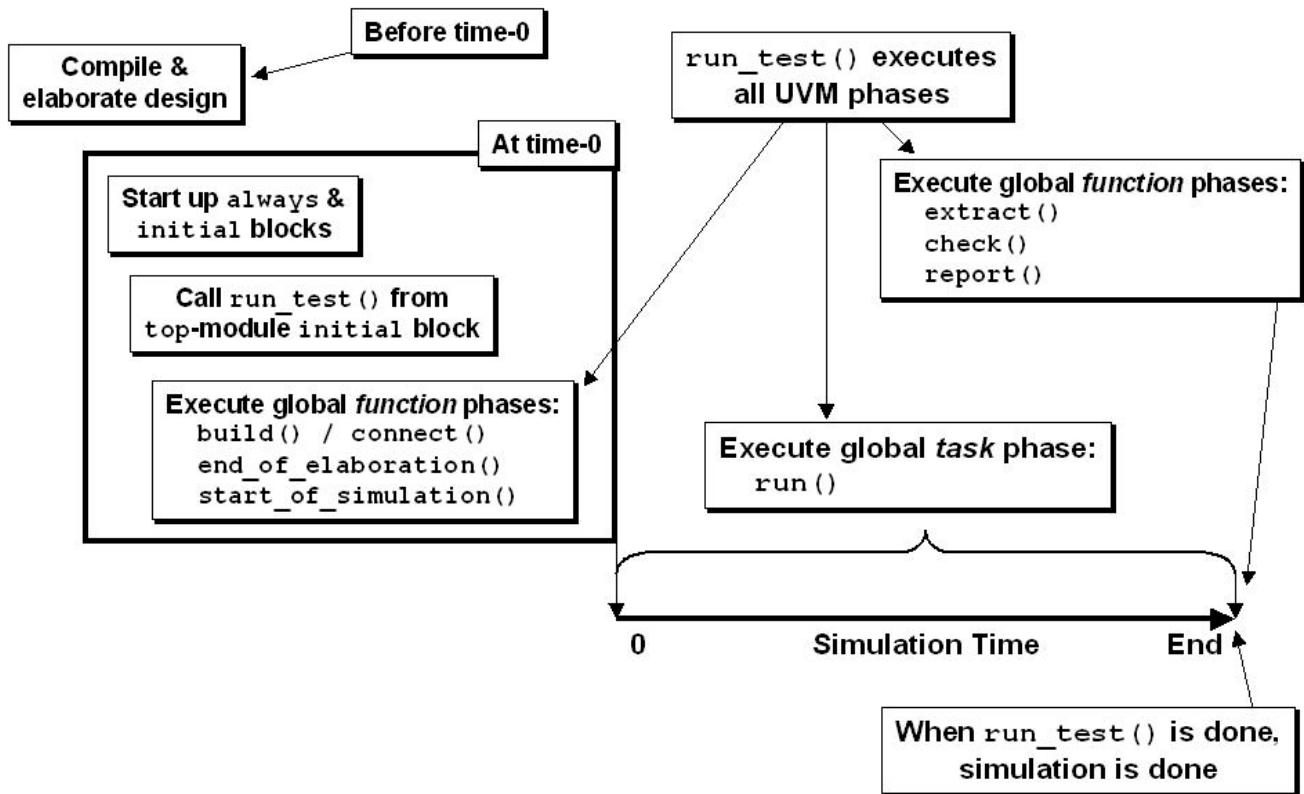


Figure 1 - (1) Compiling designs & running UVM - overview

As shown in Figure 1, a design and testbench are first compiled, then the design and testbench are elaborated. Design and elaboration happen before the start of simulation at time-0.

At time-0, the procedural blocks (**initial** and **always** blocks) in the top-level module and in the rest of the design start running. In the top-level module is an **initial** block that calls the **run\_test()** task from **uvm\_top**. When **run\_test()** is called at time-0, the UVM pre-**run()** global function phases (**build()**, **connect()**, **end\_of\_elaboration()**, **start\_of\_simulation()**) all execute and complete. After the pre-**run()** global function phases complete (still at time-0), the global **run()** phase starts. The **run()** phase is a task-based phase that executes the entire simulation, consuming all of the simulation time. When the **run()** phase stops, the UVM post-**run()** global function phases (**extract()**, **check()**, **report()**) all run in the last time slot before simulation ends.

### 3. PHASES

For discussion purposes, it is useful to start with a quick reminder of the standard UVM phases. Some brief references will be made to these phases throughout the content of this paper.

Some of the best UVM (and OVM) documentation is actually buried in the comments of the UVM base class source files themselves, if you know where to look. A concise description of the UVM phases can be found in the `uvm_phases.sv` file in the `uvm/src/base` subdirectory.

In fact, the following excellent summary-description of the UVM phases comes from comments on lines 284-316 of the `uvm_phases.sv` file:

```
// Section: Usage
//
// Phases are a synchronizing mechanism for the environment. They are
// represented by callback methods. A set of predefined phases and corresponding
// callbacks are provided in uvm_component. Any class deriving from
// uvm_component may implement any or all of these callbacks, which are executed
// in a particular order. Depending on the properties of any given phase, the
// corresponding callback is either a function or task, and it is executed in
// top-down or bottom-up order.
//
// The UVM provides the following predefined phases for all uvm_components.
//
//   build      - Depending on configuration and factory settings,
//               create and configure additional component hierarchies.
//
//   connect    - Connect ports, exports, and implementations (imps).
//
//   end_of_elaboration - Perform final configuration, topology, connection,
//               and other integrity checks.
//
//   start_of_simulation - Do pre-run activities such as printing banners,
//               pre-loading memories, etc.
//
//   run        - Most verification is done in this time-consuming phase. May fork
//               other processes. Phase ends when global_stop_request is called
//               explicitly.
//
//   extract    - Collect information from the run in preparation for checking.
//
//   check      - Check simulation results against expected outcome.
//
//   report     - Report simulation results.
```

Of these eight standard pre-defined phases, four of the phases are pre-`run()` function phases that execute in zero-time after compiling and elaborating the design (typically at time-0) but before the `run()` phase commences, and three of the phases are post-`run()` function phases that execute in zero-time at the end of the simulation after the `run()` phase completes.

#### 3.1 Run() Phase

The `run()` phase is the only standard pre-defined phase where all UVM simulation activity is executed.

Included in lines 284-425 of the 427 lines that make up the `uvm_phases.sv` file, are included brief descriptions of: a summary-description of the UVM phases (lines 284-316), discussion of the UVM phase subtype (lines 317-320), requirements to create a user-defined phase (lines 321-355) and a description of the optional Phase Macros (lines 321-425).

This paper largely deals with starting and stopping the `run()` phase. We briefly mention the pre-`run()` function phases by way of introduction, but then most of this paper talks about how the `run()` phase is started and how it is gracefully terminated. Graceful

termination of the **run()** phase allows the rest of the UVM post-**run()** function phases to do their intended jobs and then to terminate gracefully.

Graceful termination of the **run()** phase often requires the use of UVM built-in termination commands, such as **global\_stop\_request()**, and others described in this paper.

The **run()** phase is a time-consuming phase. The **run()** phase will *unconditionally* execute all of the **run** tasks, and *conditionally* all of the **stop** tasks in the UVM testbench. **run** and **stop** are empty virtual tasks defined in the **uvm\_component**. Any testbench component that is derived from **uvm\_component** can override the **run** and **stop** tasks and they will be executed during the **run()** phase either unconditionally (**run**) or conditionally (**stop**).

Some of the common UVM components that are derived from **uvm\_component** include:

Location: **uvm/src/base/**

```
uvm_root.svh:                class uvm_root
```

Location: **uvm/src/methodology/**

```
uvm_agent.svh:                virtual class uvm_agent
uvm_driver.svh:                class uvm_driver          #( ... )
uvm_env.svh:                  virtual class uvm_env
uvm_monitor.svh:              virtual class uvm_monitor
uvm_scoreboard.svh:           virtual class uvm_scoreboard
uvm_subscriber.svh:           virtual class uvm_subscriber #( ... )
uvm_test.svh:                 virtual class uvm_test
```

Location: **uvm/src/methodology/sequences/**

```
uvm_sequencer_base.svh:       class uvm_sequencer_base
```

### 3.2 Run() Phase Stages

For reference purposes in this paper, the **run()** phase has been divided into two stages and the three different types of execution threads that can run in those stages (the different threads are discussed in section 3.3). The two **run()** phase stages are the Active Stage, and the Stop-Interrupt Stage, as shown in Figure 2.

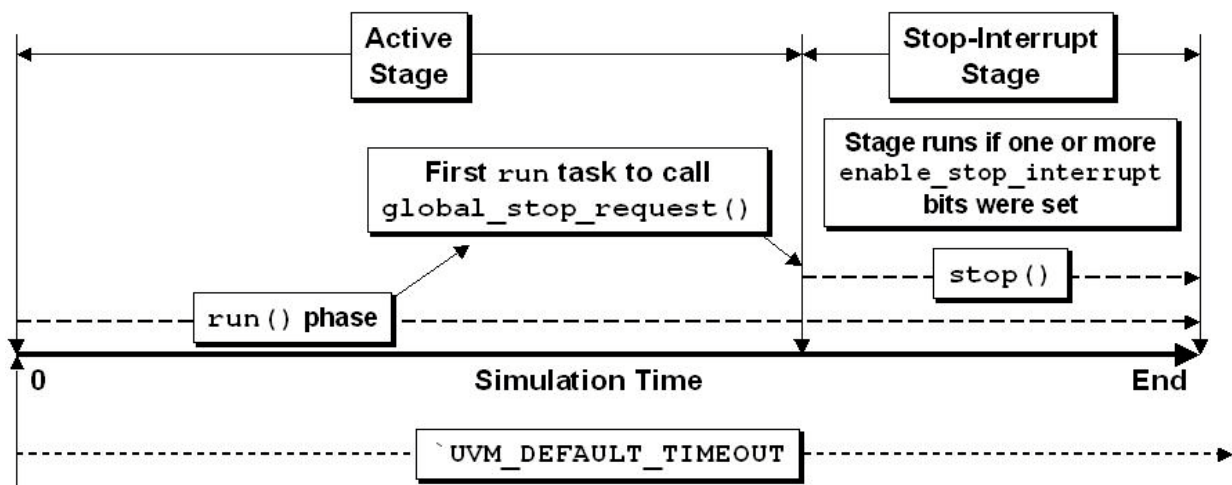


Figure 2 - **run()** phase - Active Stage, Stop-Interrupt Stage and Timeout

Execution flow of the **run** and **stop** tasks in these stages is shown in the flow diagram of Figure 3.

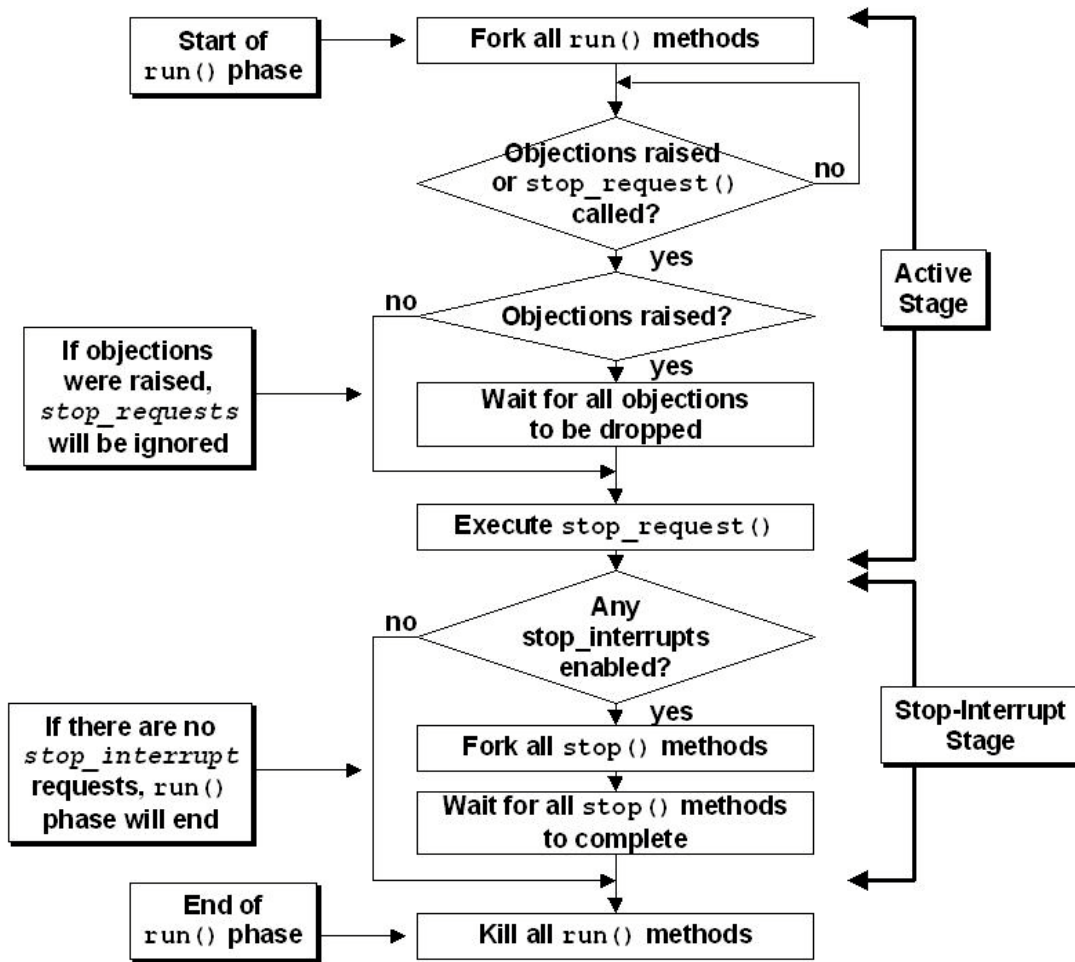


Figure 3 - run() phase execution flow diagram (without timeouts)

All **run** tasks are forked at the beginning of the Active Stage of the **run()** phase, while only components that have set the **enable\_stop\_interrupt** bit will execute the **stop** tasks during the Stop-Interrupt Stage.

More details about the execution of **run** and **stop** tasks are detailed throughout the remainder of this paper.

### 3.3 Run() Phase Threads

Again, for reference purposes in this paper, the **run()** phase execution threads can be divided into three types: *Non-Stopping* threads, *Stop-Request* threads and *Objections-Raised* threads.

All three types of execution threads are **run** tasks defined in tests or components that are directly or indirectly extended from **uvm\_component**, and all three begin execution as forked **run** tasks at the beginning of the Active Stage.

#### 3.3.1 Non-Stopping Threads

Non-Stopping threads are **run** tasks that either execute their code without issuing a stop command or are **run** tasks with a **forever** loop that never ends.

A **run** task that does not make a call to **global\_stop\_request()** or that does not raise any objections would be a non-stopping thread.

A common **run** task example that includes a **forever** loop would be a UVM testbench driver. The driver might have a **forever** loop that continuously loops and calls **seq\_item\_port.get\_next\_item(tx)** to get the next transaction that will be supplied by the UVM sequencer. Placing a **forever** loop in a driver is a common practice that makes the driver independent of the number of transactions that may be supplied by the test through the sequencer.

### 3.3.2 Stop-Request Threads

Stop-Request threads are **run** tasks that call the **global\_stop\_request()** command inside of the **run** task. If there are no Objections-Raised threads, the **run()** phase will immediately process the **stop\_request()**, terminate the Active Stage and start the Stop-Interrupt Stage if any of the **enable\_stop\_interrupt** bits were set in any of the **run()** phase threads.

If there are any Objections-Raised threads, the **global\_stop\_request()** command is largely ignored. For this reason, the **global\_stop\_request()** command is probably not the best way to terminate the Active Stage of the **run()** phase. All it takes is a single Objections-Raised thread to invalidate all of the **global\_stop\_request()** commands in all of the Stop-Request threads.

### 3.3.3 Objections-Raised Threads

Objections-Raised threads are **run** tasks that call the command **uvm\_test\_done.raise\_objection()**, typically at the beginning of the **run** task.

A **run** task that calls **uvm\_test\_done.raise\_objection()**, is specifying that it "objects to the termination of the Active Stage of the **run()** phase," until the objection is dropped using the **uvm\_test\_done.drop\_objection()** command.

Once any thread calls **uvm\_test\_done.raise\_objection()**, all calls to **global\_stop\_request()** from Stop-Request threads will be ignored. For this reason, the following guideline is recommended:

**Guideline:** Use the **raise\_objection()** / **drop\_objection()** mechanism to terminate the Active Stage of the **run()** phase.

**Reason:** Adding a single Raised-Objection thread to a test that previously used Stop-Request threads exclusively will turn off all stopping mechanisms that were present in the Stop-Request threads. Choosing Objection-Raised threads over Stop-Request threads, is a defensive coding style that will prevent confusion related to **global\_stop\_request()** commands inexplicably failing to stop the Active Stage of the **run()** phase.

Once all of the threads with raised objections have been dropped, the **run()** phase will immediately process an implicit **stop\_request()**, terminate the Active Stage and start the Stop-Interrupt Stage if any of the **enable\_stop\_interrupt** bits were set in any of the **run()** phase threads.

Dropping all objections will issue an immediate and implicit **stop\_request()**, even if there is a free-running **forever**-loop in a Non-Stopping thread.

As a general rule, issuing the **uvm\_test\_done.raise\_objection()** command in an Objections-Raised thread should be done at the beginning of the **run** task, but it can actually be issued anywhere in the **run** task as long as it is issued before the end of the Active Stage. Once the Stop-Interrupt Stage starts, calling **uvm\_test\_done.raise\_objection()** will have no effect and no objection for that thread will be raised.

A thread cannot object to the termination of the Stop-Interrupt Stage. Setting the **enable\_stop\_interrupt** bit in one or more threads is the mechanism that is used to interrupt the termination of the Stop-Interrupt Stage, as described in the next section.

### 3.3.4 Enabling Stop-Interrupts In Threads

Any of the three thread types can also delay completion of the **run()** phase by declaring their intent to interrupt the stopping of the **run()** phase.

To interrupt the stoppage of the **run()** phase requires that a thread set the **enable\_stop\_interrupt** bit, typically at the beginning of the **run** task. Once the **enable\_stop\_interrupt** bit is set, the thread will also execute a **stop** task in the Stop-Interrupt Stage. Even Non-Stopping threads with **forever** loops can set the **enable\_stop\_interrupt** bit and include a **stop** task that will execute in parallel with the thread's **forever**-loop activity in the Stop-Interrupt Stage.

As a general rule, setting the **enable\_stop\_interrupt** bit for any thread type should be done at the beginning of the **run** task, but it can actually be set anywhere in the **run** task as long as it is set before the end of the Active Stage. Once the Stop-Interrupt Stage starts, setting the **enable\_stop\_interrupt** bit will have no effect and any local **stop** tasks will be ignored.

The base class **stop()** method is defined in the **uvm\_component.sv** file as a simple empty task as shown in Example 1.

```
task uvm_component::stop(string ph_name);
    return;
endtask
```

Example 1 - uvm\_component base stop() method definition

If the thread with `enable_stop_interrupt=1` does not include a local `stop` task, then the default empty `stop()` method will execute and return in zero time.

As a general rule, threads that set the `enable_stop_interrupt` bit should also define a local `stop` task to override the default `stop()` method. The authors can think of no good reason to set the `enable_stop_interrupt` bit and omit the inclusion of a local `stop` task.

## 4. HOW UVM SIMULATIONS WORK

There are a few poorly-understood features that if generally understood would help explain how UVM simulations work.

First, how do some of the important UVM commands work and how do they become available?

Briefly, in a top-level module, an engineer:

1. Imports the `uvm_pkg`.
2. Instantiates a Design Under Test (DUT) with design interface that is used to tie the class-based testbench to the DUT.
3. Calls a UVM `run_test()` method.

Of course there is more to a UVM top-module than just these three pieces, but this will serve as a starting point.

What do we get when we `import uvm_pkg::*` ?

The `uvm_pkg.sv` is located in the `uvm/src` directory and is little more than an ``include` command enclosed within a package as shown in Example 2.

```
package uvm_pkg;
  `include "uvm.svh"
endpackage
```

Example 2 - Abbreviated `uvm_pkg.sv` file

The included `uvm.svh` file itself includes global UVM macros, and calls on three more include files located in the `uvm/base`, `uvm/uvm_tlm` and `uvm/methodology` subdirectories respectively as shown in Example 3.

```
//`include "uvm_macros.svh"
`include "base/base.svh"
//`include "uvm_tlm/uvm_tlm.svh"
//`include "methodology/methodology.svh"
```

Example 3 - Abbreviated `uvm.svh` file (with some ``include` files commented out)

In Example 3, we have commented out all of the include commands except for the `"base/base.svh"` include command, because we are going to put together a simple example to show how some important UVM simulation commands are used.

The `base.svh` file actually includes 28 other files from the `src/base` subdirectory, but we are only going to focus on two of the included files as shown in Example 4.

```
`include "base/uvm_component.sv"
//-----
// uvm_component includes uvm_root
//-----
`include "base/uvm_globals.svh"
```

Example 4 - Abbreviated `base/base.svh` file (shows inclusion of 2 of 28 included files)

The `uvm/base` subdirectory actually contains 39 files, but some of the files in this subdirectory include other files in the same subdirectory. In fact, as indicated in Example 4, the `uvm_component.sv` file includes the very important `uvm_root.svh` file, which is discussed next.

### 4.1 uvm\_root

From comments in the `uvm_root.svh` source code file[11] (also in the `ovm_root.svh` source code file[8]):

"The `uvm_root` class serves as the implicit top-level and phase controller for all UVM components. Users do not directly instantiate `uvm_root`. The UVM automatically creates a single instance of `<uvm_root>` that users can access via the global (`uvm_pkg`-scope) variable, `uvm_top`."

Among other things, the `uvm_root.svh` file, contains the `uvm_root` class definition, which is an extension of `uvm_component`. `uvm_root` also has some very important methods and variables that are used at the beginning of the simulation. Important pieces of an abbreviated `uvm_root` file with line numbers are shown in Example 5 and will be discussed in more detail in this section.

```

1 class uvm_root extends uvm_component;
2   extern protected function new (); // Later line
3
4   extern static function uvm_root get();
5
6   extern virtual task run_test (string test_name="");
7
8   extern task run_global_phase ();
9
10  extern function void stop_request();
11
12  time phase_timeout = 0;
13  time stop_timeout  = 0;
14
15  //extern `protected function new ();
16
17  static local uvm_root m_inst;
18
19  function void uvm_rocks ();
20    $display("\n*** UVM rocks! - This is the future of verification! ***\n");
21  endfunction
22 endclass
23
24 //-----
25 // IMPLEMENTATION
26 //-----
27
28 function uvm_root uvm_root::get();
29   if (m_inst == null)
30     m_inst = new();
31   return m_inst;
32 endfunction
33
34 function uvm_root::new();
35 endfunction
36
37 //-----
38 // Create a top-level handle called: uvm_top
39 //-----
40
41 const uvm_root uvm_top = uvm_root::get();
42
43 //-----
44 // Primary Simulation Entry Points
45 //-----
46
47 task uvm_root::run_test(string test_name="");
48   bit testname_plusarg;
49   string msg;
50
51   testname_plusarg = 0;
52
53   // plusarg overrides argument
54   if ($value$plusargs("UVM_TESTNAME=%s", test_name))
55     testname_plusarg = 1;
56
57   // if test_name is not "", create it using common factory
58   if (test_name != "") begin

```



```

59     msg = testname_plusarg ? "command line +UVM_TESTNAME=":
60                               "call to run_test(";
61     $display("INVTST",
62             {"Requested test from ",msg, test_name, ") not found." });
63     $finish;
64 end
65
66 $display("RNTST", {" Running test ",test_name, "..."});
67
68 run_global_phase();
69 endtask
70
71 task uvm_root::run_global_phase();
72     $display("run_global_phase() Phases now running");
73 endtask
74
75
76 //-----
77 // Stopping
78 //-----
79
80 function void uvm_root::stop_request();
81     // ->m_stop_request_e;
82     $display("Executing stop_request();");
83 endfunction

```

Example 5 - Abbreviated base/uvm\_root.svh file

The **uvm\_root** class includes the following important **extern** method definitions: **protected function new()**, **get()**, **run\_test()**, **run\_global\_phase()**, and **stop\_request()**. The external definitions are actually included later in the same file (for example, the **task uvm\_root::run\_test(...)**; definition starts on line 47). In fact *all* of the definitions for the **extern** methods are actually defined later in the same file, which means they are not very **extern-al**! So why declare all of these methods as **extern** methods? By declaring all of the methods as **extern** methods, the first 240 lines of the **uvm\_root.svh** file serve as documentation for and prototypes of the methods and important variables that make up the **uvm\_root** non-virtual class. The details of how the methods work are included in lines 266-1218 of the rest of this file. When you open the file, the top portion has an explanation of what you will find in the **uvm\_root** class and the bottom portion has the implementation details.

The **uvm\_root** class also includes definitions for the **time** variables **phase\_timeout**, **stop\_timeout** and the **static local m\_inst** handle of type **uvm\_root**.

This abbreviated version of **uvm\_root** has been augmented with the **function void uvm\_rocks()** method, which is not in the actual **uvm\_root.svh** file, but has been included in this abbreviated file for demonstration purposes as shown Example 6.

## 4.2 uvm\_root Typical Use

Before talking about the **base/uvm\_globals.svh** file, it is instructive to look at how the **uvm\_root** file is used in a typical verification environment.

The top-level module, calls the **import uvm\_pkg::\*;** command. **uvm\_pkg** includes **uvm.svh**, which includes **base/base.svh**, which includes **base/uvm\_component.sv**, which includes **base/uvm\_root.svh**.

The inclusion of **uvm\_root.svh** is where UVM gets interesting. The following discussion will refer to the abbreviated **uvm\_root.svh** line numbers shown in Example 5.

On line 2, there is an **extern protected function new()**. This line does not exist in the actual **uvm\_root.svh** file, but the commented out **extern `\_\_protected function new()** on line 15 does, and accomplishes the same goal. Declaring the **new()** constructor as **protected** means that only methods inside of **uvm\_root** can call the constructor. **uvm\_root** cannot be declared and constructed outside of **uvm\_root**.

On line 17 is the declaration of a **local static** handle called **m\_inst** of type **uvm\_root**. On line 4 is the **extern static function uvm\_root get()**; declaration, and on lines 28-32 is the external definition of the **get()** function. The first time that **uvm\_root::get()** is called, the **m\_inst** handle will be null so the **get()** function will call the protected **new()** constructor to

create a **uvm\_root** object with handle name **m\_inst** and return the handle to the calling code. All subsequent calls to **get()** will just return the **static m\_inst** handle value.

```
17 static local uvm_root m_inst;

4  extern static function uvm_root get();

28 function uvm_root uvm_root::get();
29   if (m_inst == null)
30     m_inst = new();
31   return m_inst;
32 endfunction
```

In the same **uvm\_root.svh** file is a declaration for a **const uvm\_root uvm\_top** (a constant handle of type **uvm\_root** with handle name **uvm\_top**) that calls **uvm\_root::get()** in its declaration.

```
41 const uvm_root uvm_top = uvm_root::get();
```

This is how the **uvm\_top** is created and it is all done by importing the **uvm\_pkg** in the top-level module and the construction of **uvm\_top** happens right after simulation starts at time 0. **uvm\_top** is constructed before we even execute the UVM phases; in fact, the UVM phases are executed by calling the **run\_test()** method from this instance (object) of **uvm\_top**. Now **uvm\_top** is a fully constructed object of type **uvm\_root**, and it is now possible to call any **uvm\_root** method (including **run\_test()**) just by using the **uvm\_top** handle.

Consider the simple test module example in Example 6. After importing **uvm\_pkg**, although we have not directly called a constructor to create a **uvm\_top** handle of type **uvm\_root**, we can still call the **uvm\_rocks()** method (shown in Example 7) using the **uvm\_top** handle.

```
module test;
  import uvm_pkg::*; // import uvm base classes
  initial begin
    uvm_top.uvm_rocks();
  end
endmodule
```

Example 6 - Simple test module example that calls the **uvm\_top.uvm\_rocks()** method

```
19  function void uvm_rocks ();
20    $display("\n*** UVM rocks! - This is the future of verification! ***\n");
21  endfunction
```

Example 7 - **uvm\_rocks()** method definition

The **uvm\_rocks()** method shown in Example 7 is our simple replacement for the ubiquitous C-language "hello world" example (*which is so passé!*)

A more typical usage example is a top-level module that calls a **run\_test()** method as shown in Example 8.

```
module top;
  import uvm_pkg::*; // import uvm base classes

  initial begin
    run_test();
  end
endmodule
```

Example 8 - Top module example with **run\_test()** call

### 4.3 run\_test()

The `run_test()` method is also defined in the `uvm_root` class. Most `initial` block calls to `run_test()` do not reference this method with the `uvm_top` handle name. Why does this work?

Now examine the `base/uvm_globals.svh` file. An abbreviated version of the `uvm_globals.svh` file is shown in Example 9.

```
//-----  
// Group: Simulation Control  
//-----  
task run_test (string test_name="");  
    uvm_root top;  
    top = uvm_root::get();  
    top.run_test(test_name);  
endtask  
  
function void global_stop_request();  
    uvm_root top;  
    top = uvm_root::get();  
    top.stop_request();  
endfunction  
  
function void set_global_timeout(time timeout);  
    uvm_root top;  
    top = uvm_root::get();  
    top.phase_timeout = timeout;  
endfunction  
  
function void set_global_stop_timeout(time timeout);  
    uvm_root top;  
    top = uvm_root::get();  
    top.stop_timeout = timeout;  
endfunction
```

Example 9 - Abbreviated `base/uvm_globals.svh` file

The tasks and functions in the `uvm_globals.svh` file are also in the `uvm_pkg`, but they are not part of any class definition.

A call to `run_test()`, as shown in Example 8, calls the `run_test()` task defined in Example 9. The `run_test()` task declares a handle called `top` of the `uvm_root` class type. Then the `uvm_root::get()` method is called, which will return the **local static** `uvm_root` handle, `m_inst`, and store it into the `top` handle declared in the task (the `uvm_root::get()` method ensures that only one `uvm_root` object will ever be created), and using the `top` handle, the local `run_test()` calls the `top.run_test()` method in the `uvm_top` object, which starts up all the UVM simulation phases.

The `uvm_globals.svh` file also contains the `set_global_timeout()` and `set_global_stop_timeout()` methods, which will be discussed in section 6.1.

As a side note, although not shown in the abbreviated version of the `uvm_globals.svh` file in Example 9, the `uvm_globals.svh` file also contains the UVM standard message commands: `uvm_report_info()`, `uvm_report_warning()`, `uvm_report_error()` and `uvm_report_fatal()`. These commands are also available anywhere the `uvm_pkg` routines have been imported.

To summarize what has been discussed in this section, `import uvm_pkg::*;` creates the `uvm_top` module and includes important global commands such as `run_test()`, `global_stop_request()`, `set_global_timeout()` and `set_global_phase_timeout()`. Users of the `uvm_pkg` have access to commands without being required to construct any class objects.

## 5. HOW UVM SIMULATIONS RUN

As mentioned in the previous section, the `run_test()` command starts all of the UVM phases. This section gives more details on what happens when simulations run.

### 5.1 Choosing A Test To Run

The `run_test()` command must be passed a valid test name that has been registered in the UVM factory. There are two ways to pass a valid test name to the `run_test()` command, (1) coded into the `top` module or (2) passed to the UVM testbench through the command line switch `+UVM_TESTNAME`.

The inline coded method passes the test name string as an argument to the `run_test()` method in the top-level module, similar to what is shown below.

```
module top;
  ...

  initial begin
    run_test("test1");
    ...
  end
endmodule
```

The inline coded method is not typically recommended since it ties the testbench to a specific test that requires the `top`-module to be modified, recompiled and simulated for each new test.

The `+UVM_TESTNAME` command line switch is the preferred method for executing tests since the simulation executable can be called with a new testname without the requirement to re-compile the entire testbench. Below is shown an example command line switch using Questa:

```
vsim -c -do "run -all" top +UVM_TESTNAME=test1
```

### 5.2 Common +UVM\_TESTNAME Errors

If no top-module `run_test()` argument is included in the `top`-module, and if there is no `+UVM_TESTNAME` command line argument, UVM reports the error shown in Figure 4.

```
UVM_FATAL @ 0: reporter [NOCOMP] No components instantiated.
You must instantiate at least one component before calling run_test.
To run a test, use +UVM_TESTNAME or supply the test name in the
argument to run_test(). Exiting simulation.
```

Figure 4 - UVM\_FATAL - No components instantiated - missing test name

If the `run_test()` test name argument included in the `top`-module has not been registered in the UVM factory, or if the `+UVM_TESTNAME` test name has not been registered in the UVM factory, then UVM reports the error shown in Figure 5.

```
UVM_WARNING @ 0: reporter [BDTYP] Cannot create a component of type
'test1' because it is not registered with the factory.
UVM_FATAL @ 0: reporter [INVTST] Requested test from command line
+UVM_TESTNAME=test1 not found.
```

Figure 5 - UVM\_FATAL - +UVM\_TESTNAME not found - test not registered with the factory

### 5.3 Verilog-Style Testbench

Before looking at common UVM testbench styles, consider how Verilog verification engineers typically build a testbench.

Verilog tasks are encapsulated, time-consuming subroutines that execute their code when called and return when done. The experienced Verilog verification engineer typically assembles a large number of tasks and then makes calls to the tasks from a Verilog testbench. The tasks are used as high level commands that either execute important Verilog testbench activity or are used to apply stimulus to, or check outputs from a Verilog DUT.

At the end of the test, after all of the testing tasks have run, the typical verification engineer terminates the test with a call to **\$finish**; as shown in Example 10.

```
module top;
  import uvm_pkg::*;
  logic      clk;
  clkgen     ck  (clk);

  initial begin
    run_task();
    $finish;
  end

  task run_task;
    uvm_report_info("top","run_task running");
    #100ns;
  endtask
endmodule
```

Example 10 - Verilog style testbench with task call followed by \$finish

Since the **uvm\_pkg** was imported in Example 10, the **top** module has access to the UVM reporting commands, which happened to be called from the Verilog testbench **run\_task**. Even though the **uvm\_pkg** was imported, it was not necessary to call the UVM **run\_test()** command, so none of the UVM phases were executed, including the final reporting phase, and the final output simply shows that **UVM\_INFO** was called and that the **top** module finished at time 100ns, as shown in Figure 6.

```
UVM_INFO @ 0: reporter [top] run_task running
** Note: $finish ...
Time: 100 ns ...
```

Figure 6 - Verilog style testbench output

### 5.4 Common New-User UVM Testbench Coding Mistake

Based on the previous Verilog coding style, the novice UVM verification engineer frequently follows the Verilog style by calling the **run\_test()** command followed immediately by calling a **global\_stop\_request()** command, as shown in Example 11.

```
module top;
  import uvm_pkg::*; // import ovm base classes
  import tb_pkg::*;  // import testbench classes
  logic      clk;
  clkgen     ck  (clk);

  initial begin
    run_test();
    global_stop_request();
  end
endmodule
```

Runs all UVM phases so  
**global\_stop\_request()**  
will never execute

Example 11 - BAD - top module with common mistake - **global\_stop\_request()** after **run\_test()**

Engineers who assemble testing code as shown in Example 11 fail to understand that the `run_test()` command will execute all of the UVM phases and the placement of the `global_stop_request()` command immediately after the `run_test()` command is too late to be included in the execution of any of the UVM phases. In this example, the only call to `global_stop_request()` is placed in the `top`-module, so the `run_test()` command will execute the requested `run()` phase `test1` code, shown in Example 12, then keep executing simulation in the `run()` phase until simulation times out with the error message shown in Figure 7.


```
class test1 extends uvm_test;
  `uvm_component_utils(test1)
  env e;

  function new (string name="test1", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    #100ns;
  endtask
endclass
```

`run()` task executes a 100ns delay but does not include a `global_stop_request()`



Example 12 - test1 with NO `global_stop_request()`

```
UVM_INFO @ 0: reporter [RNTST] Running test test1...
UVM_ERROR @ 92000000000000: reporter [TIMEOUT] Watchdog timeout
of '92000000000000' expired
```

Figure 7 - Test timeout - BAD `global_stop_request()` after `run_test()` command

The `test1` code of Example 12 was run with the `top`-module shown in Example 11, but since the `test1` code did not include a `global_stop_request()` command, and since the `global_stop_request()` command of the `top` module was never executed, `test1` executed its code and then the `run()` phase continued until the simulation timed-out.

Placement of a `global_stop_request()` command after a call to `run_test()` is a Verilog-like coding style that does not work in UVM.

## 5.5 Proper Use of `global_stop_request()` Command

The proper use of the `global_stop_request()` command is to omit it from the `top` module as shown in Example 13, and to include the `global_stop_request()` command in the test code as shown in Example 14.

```
module top;
  import uvm_pkg::*;
  import tb_pkg::*;

  logic          clk;
  clkgen         ck  (clk);

  initial begin
    run_test();
  end
endmodule
```

Example 13 - GOOD - `top` module with `run_test()` and NO `global_stop_request()`

```

class test2 extends uvm_test;
  `uvm_component_utils(test2)
  env e;

  function new (string name="test2", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    #100ns;
    global_stop_request();
  endtask
endclass

```

**run()** task executes a 100ns delay but followed by a call to **global\_stop\_request()**

Example 14 - GOOD - test2 terminates with global\_stop\_request()

The **top**-module of Example 13 still includes the **run\_test()** command, which is responsible for initiating and executing all of the UVM phases.

After the **test2** code of Example 14 terminates with **global\_stop\_request()**, the rest of the post-**run()** phases will execute (as shown in Figure 8) and the full UVM report-output will be displayed as shown in Figure 9.

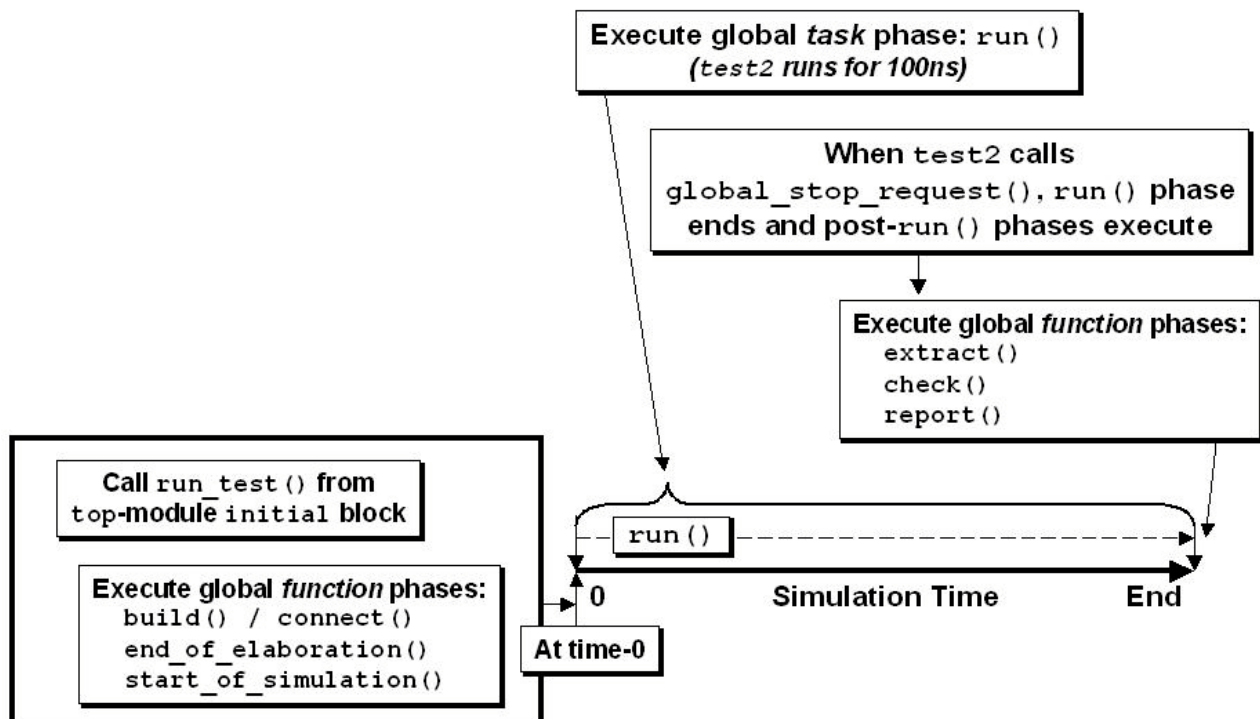


Figure 8 - global\_stop\_request() terminates the run() phase and post-run() phases execute

```

UVM_INFO @ 0: reporter [RNTST] Running test test2...

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    1
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]      1
** Note: $finish ...
Time: 100 ns ...

```

Figure 9 - UVM report output after proper termination of the `run()` phase using call to `global_stop_request()`

If the `global_stop_request()` command in the test code is replaced with the Verilog `$finish` command as shown in Example 15, the simulation will abort in the middle of the `run()` phase and the post-`run()` phases will not execute as shown in Figure 10. Since the UVM phases abort in the middle of the `run()` phase, no final UVM reports will be printed as can be seen in the resultant output in Figure 11.

```

class test2a extends uvm_test;
  `uvm_component_utils(test2a)
  env e;

  function new (string name="test2a", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    #100ns;
    $finish;
  endtask
endclass

```

Example 15 - Bad - test2a terminates with `$finish`; - `run()` phase aborts early



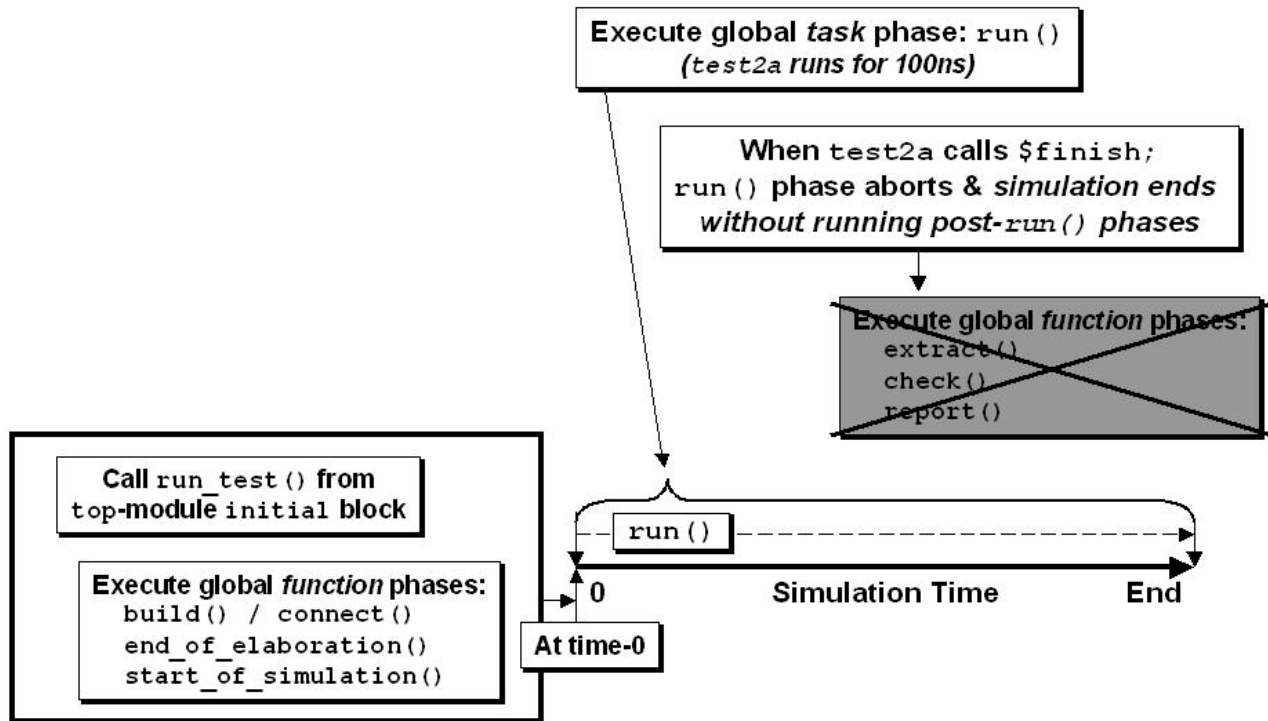


Figure 10 - \$finish command causes the run() phase to abort and post-run() phases never execute

```
UVM_INFO @ 0: reporter [RNTST] Running test test2a...
** Note: $finish      : test2a.sv ...
Time: 100 ns Instance: /uvm_pkg::uvm_root::m_do_phase
```

Figure 11 - \$finish; causes simulation to abort run() phase - no final reports printed

In Figure 11, there is no UVM Report Summary because the post-run() **report()** phase was never executed.

## 6. HOW UVM SIMULATIONS STOP

UVM has a number of important, built-in global simulation and control commands to set timeouts and to terminate the **run()** phase. Descriptions of these UVM commands can be found in the UVM Class Reference manual[9] (also in the OVM Class Reference manual[6]).

**run\_test()** - is a commonly used convenience function that calls **uvm\_top.run\_test()**. It is used to start the execution of the specified test.

**global\_stop\_request()** - is a commonly used convenience function that calls **uvm\_top.stop\_request()**. It is used to terminate the Active Stage if there are no raised objections (see Objections-Raised Threads in section 3.3.3)

**set\_global\_timeout()** - is a convenience function that calls **uvm\_top.phase\_timeout = timeout**.

**set\_global\_stop\_timeout()** is a convenience function that calls **uvm\_top.stop\_timeout = timeout**.

The basic UVM simulation termination command is the **global\_stop\_request()** call as previously shown in Example 14.

Two other ways to terminate simulations are timeouts and a combination of raised-objections and stop-interrupts.

### 6.1 Timeouts

When the **run()** phase starts, a parallel timeout timer is also started. If the timeout timer reaches one of the specified timeout limits before the **run()** phase completes, the **run()** phase will timeout and:

1. All **run** tasks will be immediately disabled.
2. A timeout message will be issued.
3. Execution of the post-**run()** phases will begin.

There are two timeout counters that may become active during the **run()** phase and their timeout limits are kept in the variables **uvm\_top.phase\_timeout** and **uvm\_top.stop\_timeout**.

The **phase\_timeout** is the time that the entire **run()** phase (Active Stage and Stop-Interrupt Stage) is allowed to run before timing out. The **phase\_timeout** is often referred to as the **global\_timeout** limit as shown in Figure 12. If the **phase\_timeout** time limit is reached, a **UVM\_ERROR** will be reported.

The default value for the **phase\_timeout** limit is set from the **UVM\_DEFAULT\_TIMEOUT** macro and has a default timeout value of 9200 seconds. This value can be shortened by using the **set\_global\_timeout()** command.

As part of the **run()** phase, various components of a test might execute **stop** tasks in the Stop-Interrupt Stage. The maximum execution time of the **stop** tasks is stored in the **stop\_timeout** limit and can be controlled by the **set\_global\_stop\_timeout()** command. The default **stop\_timeout** value is also 9200 seconds. The **stop\_timeout** is often referred to as the **global\_stop\_timeout** limit as shown in Figure 12.

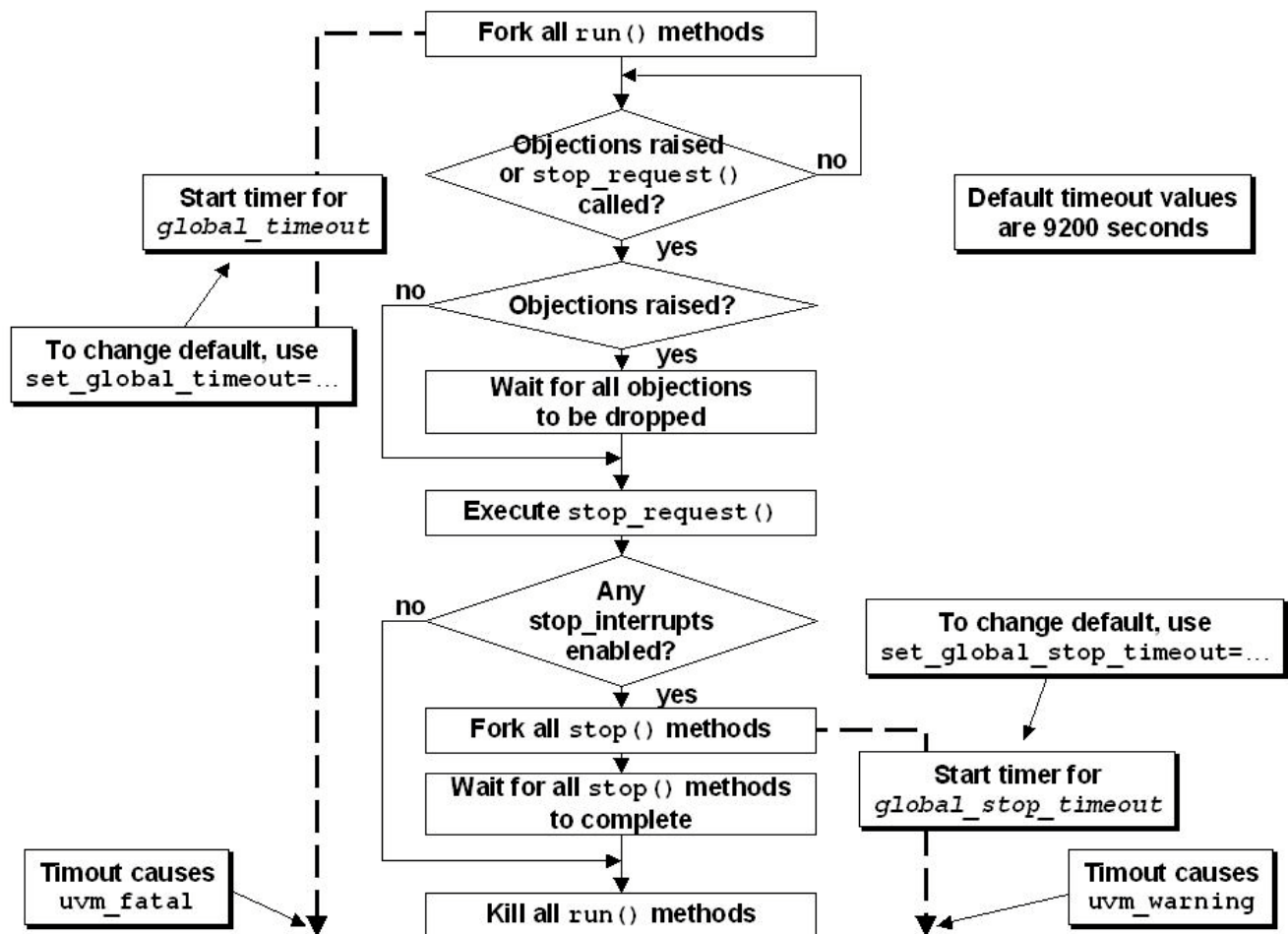


Figure 12 - `run()` phase execution flow diagram (with timeouts)

## 6.2 Stop-Interrupt Enabled Example

The `test3` code in Example 16 demonstrates the use of the `enable_stop_interrupt` bit and a local `stop` task.

A test or component (thread) can interrupt the scheduled stopping of the `run()` phase by setting the `enable_stop_interrupt` bit at the beginning of the `run` task in the test. Any thread that sets this bit should also override the default `stop` task with functionality that shall be executed in the Stop-Interrupt Stage before this `run()` phase finishes.

The `run` task in `test3`, shown in Example 16, has set the `enable_stop_interrupt` bit. Any thread that has this bit set will run the `stop` task after a `stop_request` is called. Whenever `enable_stop_interrupt=1`, the same thread typically includes a local `stop` task that has important commands that should be executed before the end of the simulation, and these commands typically take additional simulation time before the `run()` phase is allowed to finish.

When `test3` runs, the following sequence of actions will be executed:

- `run()` phase starts at time 0ns (start of Active Stage).
- `test3` executes:
  - `run` task (starts running at time 0ns)
    - `enable_stop_interrupt` bit will be set (indicates that the `test3 stop` task should interrupt and execute its code before the end of the `run()` phase).
    - Delays for 100ns.
    - Call the `global_stop_request()` command (causes end of Active Stage and start of Stop-Interrupt Stage).
  - `stop` task (starts running at time 100ns):
    - print a "stop task running" message.
    - delays for 100ns.
    - print a "stop task done" message.
- `run()` phase finishes at time 200ns (end of both Stop-Interrupt Stage and `run()` phase).

```
class test3 extends uvm_test;
  `uvm_component_utils(test3)
  env e;

  function new (string name="test3", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    enable_stop_interrupt = '1;
    #100ns;
    global_stop_request();
  endtask

  task stop (string ph_name);
    uvm_report_info("test3", "stop task running ...");
    #100ns;
    uvm_report_info("test3", "stop task done");
  endtask
endclass
```

Example 16 - test3 with enabled stop task

The printed messages and **test3** UVM Report summary are shown in Figure 13. The first printed message came from the **run\_test()** command itself, the next two printed messages came from the **stop** task in Example 16, and the UVM Report Summary came from the UVM **report()** phase at the end of the simulation.

```

UVM_INFO @ 0: reporter [RNTST] Running test test3...
UVM_INFO @ 100: uvm_test_top [test3] stop task running ...
UVM_INFO @ 200: uvm_test_top [test3] stop task done

--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :      3
UVM_WARNING :    0
UVM_ERROR :     0
UVM_FATAL :     0
** Report counts by id
[RNTST]      1
[test3]      2
** Note: $finish ...
Time: 200 ns ...

```

Figure 13 - test3 UVM reported output

The test in Example 16 executed the **run** task code in the Active Stage of the **run()** phase, while the **stop** task code executed in the Stop-Interrupt Stage of the **run()** phase as shown in Figure 14.

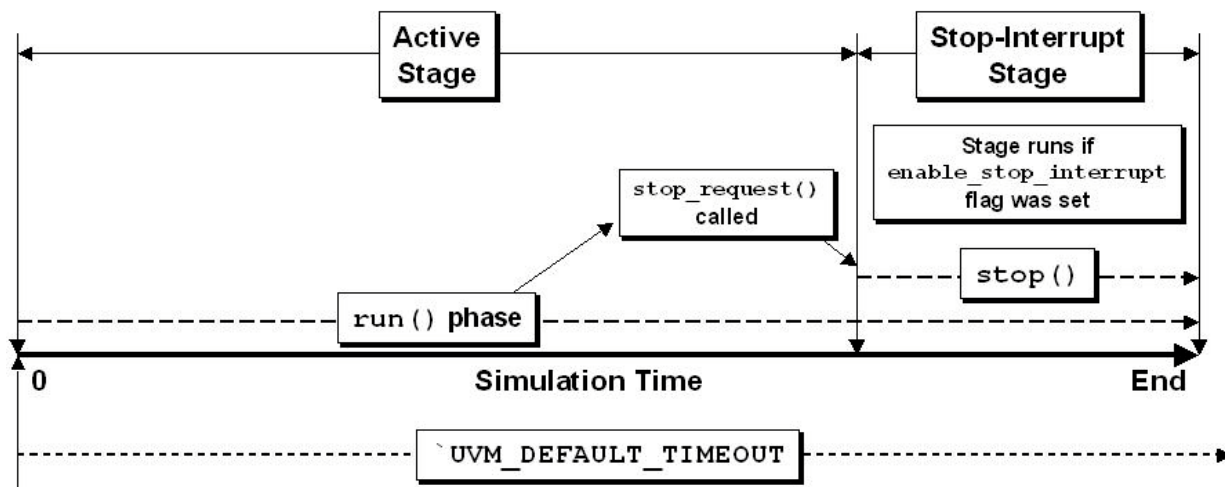


Figure 14 - (1) run() phase - Active Stage and Stop-Interrupt Stage

### 6.3 Stop\_Timeout Example

The **test4** code in Example 18 demonstrates the use of the **set\_global\_stop\_timeout()** command in the **top**-level module and the **stop** task timing out during execution of code during the Stop-Interrupt Stage.

The **top** module in Example 17 has issued the **set\_global\_stop\_timeout(50ns)** command, which means that if any thread executes code during the Stop-Interrupt Stage, the code must complete in less than 50ns otherwise a Stop-request timeout warning will be reported. **test4** will exceed the **stop\_timeout** limit and the warning will be reported.

As was done in the Stop-Interrupt Enabled example, the **run** task in **test4**, shown in Example 18, has set the **enable\_stop\_interrupt** bit and a local **stop** task includes a pair of **uvm\_report\_info** commands separated by a 100ns delay.

When **test4** runs, the following sequence of actions will be executed:

- **top** module executes (starts running at time 0ns):
  - **initial** block starts.
  - Call the **set\_global\_stop\_timeout(50ns)** command.
  - Call the **run\_test()** command.
- **run()** phase starts at time 0ns (start of Active Stage).
- **test4** executes:
  - **run** task (starts running at time 0ns)
    - **enable\_stop\_interrupt** bit will be set.
    - Delays for 100ns.
    - Call the **global\_stop\_request()** command (causes end of Active Stage and start of Stop-Interrupt Stage) .
  - **stop** task (starts running at time 100ns):
    - print a "stop task running" message.
    - delays for 100ns.
    - **UVM\_WARNING** - Stop-request timeout (reported at 150ns)
- **run()** phase times out at time 150ns (end of both Stop-Interrupt Stage and **run()** phase).

```
module top;
  import uvm_pkg::*;
  import tb_pkg::*;

  logic          clk;
  clkgen         ck  (clk);

  initial begin
    set_global_stop_timeout(50ns);
  end

  initial begin
    run_test();
  end
endmodule
```

Example 17 - top module - set\_global\_stop\_timeout()

```

class test4 extends uvm_test;
  `uvm_component_utils(test4)
  env e;

  function new (string name="test4", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    enable_stop_interrupt = '1;
    #100ns;
    global_stop_request();
  endtask

  task stop (string ph_name);
    uvm_report_info("test4", "stop task running ...");
    #100ns;
    uvm_report_info("test4", "stop task done");
  endtask
endclass

```

Example 18 - test4 - stop task timeout

The printed messages are shown in Figure 15 (UVM Report summary not shown). The first printed message came from the **run\_test()** command itself, the next printed message came from the **stop** task in Example 18, and the **UVM\_WARNING** timeout message came from the phase execution mechanism in **uvm\_root**. The UVM Report Summary (not shown) came from the UVM **report()** phase at the end of the simulation.

```

UVM_INFO @ 0: reporter [RNTST] Running test test4...
UVM_INFO @ 100: uvm_test_top [test4] stop task running ...
UVM_WARNING @ 150: reporter [STPTO] Stop-request timeout
of 50 expired. Stopping phase 'run'

```

Figure 15 - test4 UVM report output after Stop-request timeout

The **test4** example executed the **run** task code in the Active Stage and the **stop** task code in the Stop-Interrupt Stage before ending early with a UVM stop\_timeout warning also executed in the Stop-Interrupt Stage.

## 6.4 Global Timeout Example

The **test5** code in Example 20 demonstrates the use of the **set\_global\_stop\_timeout()** command in the **top**-level module and the **stop** task timing out during execution of code during the Stop-Interrupt Stage.

The **top** module in Example 19 has issued the **set\_global\_stop\_timeout(100ns)** command, which means that if any thread executes code during the Stop-Interrupt Stage, the code must complete in less than 100ns otherwise a Stop-request timeout warning will be reported. The **top** module also issued the **set\_global\_timeout(550ns)** command, which means that all the code must complete in less than 550ns otherwise a watchdog timeout (global\_timeout) error will be reported. **test5** will exceed the global\_timeout limit and the error will be reported.

When **test5** runs, the following sequence of actions will be executed:

- **top** module executes (starts running at time 0ns):
  - **initial** block starts.
    - Call the **set\_global\_stop\_timeout(100ns)** command.
    - Call the **set\_global\_timeout(550ns)** command.
    - Call the **run\_test()** command.
  - **run()** phase starts at time 0ns (start of Active Stage).
- **test5** executes:
  - **run** task (starts running at time 0ns)
    - **enable\_stop\_interrupt** bit will be set.
    - Delays for 500ns.
    - Call the **global\_stop\_request()** command (causes end of Active Stage and start of Stop-Interrupt Stage).
  - **stop** task (starts running at time 500ns):
    - print a "stop task running" message.
    - delays for 200ns.
    - **UVM\_ERROR** - global\_timeout (reported at 550ns)
- **run()** phase times out at time 550ns (end of both Stop-Interrupt Stage and run() phase).

```
module top;
  import uvm_pkg::*;
  import tb_pkg::*;

  logic          clk;
  clkgen         ck  (clk);

  initial begin
    set_global_stop_timeout(100ns);
    set_global_timeout(550ns);
  end

  initial begin
    run_test();
  end
endmodule
```

Example 19 - top module - set\_global\_stop\_timeout() & set\_global\_timeout()



```

class test5 extends uvm_test;
  `uvm_component_utils(test5)
  env e;

  function new (string name="test5", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    enable_stop_interrupt = '1;
    #500ns;
    global_stop_request();
  endtask

  task stop (string ph_name);
    uvm_report_info("test5", "stop task running ...");
    #200ns;
    uvm_report_info("test5", "stop task done");
  endtask
endclass

```

Example 20 - test5 - times out due to set\_global\_timeout() set to 550ns

The printed messages are shown in Figure 16 (UVM Report summary not shown). The first printed message came from the **run\_test()** command itself, the next printed message came from the **stop** task in Example 20, and the **UVM\_ERROR** timeout message came from the phase execution mechanism in **uvm\_root**.

<pre> UVM_INFO @ 0: reporter [RNTST] Running test test5... UVM_INFO @ 500: uvm_test_top [test5] stop task running ... UVM_ERROR @ 550: reporter [TIMOUT] Watchdog timeout of '550' expired. </pre>
--

Figure 16 - test5 UVM report output after Watchdog (global) timeout

The **test5** example executed the **run** task code in the Active Stage and the **stop** task code in the Stop-Interrupt Stage before ending early with a UVM global\_timeout error also executed in the Stop-Interrupt Stage.

## 6.5 Delayed Global\_Timeout Example

The **test6** code in Example 22 demonstrates the use of the **set\_global\_stop\_timeout()** command in the **top**-level module when the **run\_test()** command startup is delayed in the **top**-module.

The **top** module in Example 21 has issued the **set\_global\_stop\_timeout(100ns)** and **set\_global\_timeout(550ns)** commands. In this example, the **top**-module delays for 100ns before issuing the **run\_test()** command, which means that all the code must complete in less than 100ns + 550ns otherwise a watchdog timeout (global\_timeout) error will be reported. **test6** will exceed the global\_timeout limit and the error will be reported at time 650ns.

When **test6** runs, the following sequence of actions will be executed:

- **top** module executes (starts running at time 0ns):
  - **initial** block starts.
    - Call the **set\_global\_stop\_timeout(100ns)** command.
    - Call the **set\_global\_timeout(550ns)** command.
    - Delay for 100ns then call the **run\_test()** command.
- **run()** phase starts at time 100ns (start of Active Stage).
- **test6** executes:
  - **run** task (starts running at time 100ns)
    - **enable\_stop\_interrupt** bit will be set.
    - Delays for 500ns.
    - Call the **global\_stop\_request()** command (causes end of Active Stage and start of Stop-Interrupt Stage).
  - **stop** task (starts running at time 600ns):
    - print a "stop task running" message.
    - delays for 200ns.
    - **UVM\_ERROR** - global\_timeout (reported at 650ns)
- **run()** phase times out at time 650ns (end of both Stop-Interrupt Stage and **run()** phase).

```
module top;
  import uvm_pkg::*;
  import tb_pkg::*;

  logic          clk;
  clkgen         ck  (clk);

  initial begin
    set_global_stop_timeout(100ns);
    set_global_timeout(550ns);
  end

  initial begin
    #100ns run_test();
  end
endmodule
```

Example 21 - top module - set\_global\_stop\_timeout(), set\_global\_timeout() and delayed startup

```

class test6 extends uvm_test;
  `uvm_component_utils(test6)
  env e;

  function new (string name="test6", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    enable_stop_interrupt = '1;
    #500ns;
    global_stop_request();
  endtask

  task stop (string ph_name);
    uvm_report_info("test6", "stop task running ...");
    #200ns;
    uvm_report_info("test6", "stop task done");
  endtask
endclass

```

Example 22 - test6 with run() and stop() tasks

The printed messages are shown in Figure 17 (UVM Report summary not shown). The first printed message came from the **run\_test()** command itself at time 100ns, the next printed message came 500ns later from the **stop** task in Example 22, and the **UVM\_ERROR** timeout message came from the phase execution mechanism in **uvm\_root** at time 650ns.

<pre> UVM_INFO @ 100: reporter [RNTST] Running test test6... UVM_INFO @ 600: uvm_test_top [test6] stop task running ... UVM_ERROR @ 650: reporter [TIMOUT] Watchdog timeout of '550' expired. </pre>
--

Figure 17 - test6 UVM report output after Watchdog (global) timeout

The **test6** example did not start the **run()** phase and global\_timeout counter until 100ns into the simulation, then **test6** executed the **run** task code in the Active Stage and the **stop** task code in the Stop-Interrupt Stage before ending early with a UVM global\_timeout error also executed in the Stop-Interrupt Stage.

## 6.6 run Task Sets Timeout Value Example

The **test7** code in Example 24 demonstrates the use of the **set\_global\_stop\_timeout()** and the **set\_global\_timeout()** commands in the **run** task of the **test7** code instead of in the **top**-level module.

The **top** module in Example 23 is a simple **top** module with no timeout settings and no startup delays.

The **test7** code has issued the **set\_global\_stop\_timeout(100ns)** and **set\_global\_timeout(550ns)** commands in the **run** task. The problem with setting the **global\_timeout** in the **run** task is that the timeout must be set before starting the **run** task otherwise the **global\_timeout** will not work, which is what happens in this test. After the simulation passes the 550ns time without a **global\_timeout**, it will trigger the **stop\_timeout**, which can be set from the **run** task.

When **test7** runs, the following sequence of actions will be executed:

- **run()** phase starts at time 0ns (start of Active Stage).
- **test7** executes:
  - **run** task (starts running at time 0ns)
    - Call the **set\_global\_stop\_timeout(100ns)** command.
    - Call the **set\_global\_timeout(550ns)** command (TOO LATE!)
    - Set the **enable\_stop\_interrupt** bit.
    - Delays for 500ns.
    - Call the **global\_stop\_request()** command (causes end of Active Stage and start of Stop-Interrupt Stage).
  - **stop** task (starts running at time 500ns):
    - print a "stop task running" message.
    - delays for 200ns.
    - **UVM\_WARNING** - stop\_timeout (reported at 600ns)
- **run()** phase times out at time 600ns (end of both Stop-Interrupt Stage and **run()** phase).

```
module top;
  import uvm_pkg::*;
  import tb_pkg::*;

  logic          clk;
  clkgen         ck  (clk);

  initial begin
    run_test();
  end
endmodule
```

Example 23 - Simple top module used for testing - no timeout values set and no startup delays

```

class test7 extends uvm_test;
  `uvm_component_utils(test7)
  env e;

  function new (string name="test7", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    set_global_stop_timeout(100ns);
    set_global_timeout(550ns);
    enable_stop_interrupt = '1;
    #500ns;
    global_stop_request();
  endtask

  task stop (string ph_name);
    uvm_report_info("test7", "stop task running ...");
    #200ns;
    uvm_report_info("test7", "stop task done");
  endtask
endclass

```

**Example 24 - BAD: test7 with & set\_global\_timeout() set in the run() task - too late**

The printed messages are shown in Figure 18 (UVM Report summary not shown). The first printed message came from the **run\_test()** command itself, the next printed message came 500ns later from the **stop** task in Example 24, and the **UVM\_WARNING** timeout message came from the phase execution mechanism in **uvm\_root** at time 600ns.

```

UVM_INFO @ 0: reporter [RNTST] Running test test7...
UVM_INFO @ 500: uvm_test_top [test7] stop task running ...
UVM_WARNING @ 600: reporter [STPTO] Stop-request timeout
                        of 100 expired. Stopping phase 'run'

```

**Figure 18 - test7 UVM report output after Stop-request timeout**

The **test7** example shows that the **global\_timeout** limit must be set before executing the **run** task, otherwise the setting will be too late and the global timeout counter will not start.

## 6.7 build() Phase Sets Global\_Timeout Value Example

The **test8** code in Example 25 demonstrates the use of the **set\_global\_timeout()** command in the **build()** method of the **test8** code instead of in the **run** task.

The **top** module used for this test is the same simple **top** module used with **test7** (shown in Example 23).

The **test8** code has issued the **set\_global\_stop\_timeout(100ns)** command in the **run** task and has issued the **set\_global\_timeout(550ns)** command in the **build()** method. Although this coding style is not recommended, setting the **global\_timeout** in the **build()** method allows the setting to be active before executing the **run** task; hence, the setting will be active. After the simulation reaches the 550ns time the **global\_timeout**, will trigger as expected.

When **test8** runs, the following sequence of actions will be executed:

- **test8** executes:
- **build()** phase starts at time 0ns.
  - Call the **set\_global\_timeout(550ns)** command (ACTIVE!)
- **run()** phase starts at time 0ns (start of Active Stage).
  - **run** task (starts running at time 0ns)
    - Call the **set\_global\_stop\_timeout(100ns)** command.
    - Set the **enable\_stop\_interrupt** bit.
    - Delays for 500ns.
    - Call the **global\_stop\_request()** command (causes end of Active Stage and start of Stop-Interrupt Stage).
  - **stop** task (starts running at time 500ns):
    - print a "stop task running" message.
    - delays for 200ns.
    - **UVM\_ERROR** - stop\_timeout (reported at 550ns)
- **run()** phase times out at time 550ns (end of both Stop-Interrupt Stage and **run()** phase).

```
class test8 extends uvm_test;
  `uvm_component_utils(test8)
  env e;

  function new (string name="test8", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
    set_global_timeout(550ns);
  endfunction

  task run;
    set_global_stop_timeout(100ns);
    enable_stop_interrupt = '1;
    #500ns;
    global_stop_request();
  endtask

  task stop (string ph_name);
    uvm_report_info("test8", "stop task running ...");
    #200ns;
    uvm_report_info("test8", "stop task done");
  endtask
endclass
```

Example 25 - test8 with **set\_global\_timeout()** set in the **build()** method - stop-timeout is active

The printed messages are shown in Figure 19 (UVM Report summary not shown). The first printed message came from the `run_test()` command itself, the next printed message came 500ns later from the `stop` task in Example 25, and the `UVM_ERROR` timeout message came from the phase execution mechanism in `uvm_root` at time 550ns.

```
UVM_INFO @ 0: reporter [RNTST] Running test test8...
UVM_INFO @ 500: uvm_test_top [test8] stop task running ...
UVM_ERROR @ 550: reporter [TIMOUT] Watchdog timeout of '550' expired.
```

Figure 19 - test8 UVM report output after Watchdog (global) timeout

The `test8` example shows that the `global_timeout` limit can be properly set in the `build()` phase, before executing the `run` task in the `run()` phase. Although this style works, this is not necessarily a recommended coding style.

## 6.8 Raising And Dropping Objections Example

The `test9` code in Example 26 demonstrates the raising and dropping of objections in the `run` task by using the commands `uvm_test_done.raise_objection()` / `uvm_test_done.drop_objection()` respectively.

The `top` module used for this test is the same simple `top` module used with `test7` (shown in Example 23).

The `test9` code has issued the `uvm_test_done.raise_objection()` / `uvm_test_done.drop_objection()` commands in the `run` task. Once an objection is raised, all objections must be dropped before an implicit `global_stop_request()` will force the Active Stage to complete. In `test9`, there is only one raised objection so once that objection is dropped, the Active Stage finishes.

When `test9` runs, the following sequence of actions will be executed:

- `test9` executes:
- `run()` phase starts at time 0ns (start of Active Stage).
  - `run` task (starts running at time 0ns)
    - Call the `uvm_test_done.raise_objection()` command.
    - Delays for 500ns.
    - Call the `uvm_test_done.drop_objection()` command. All objections have been dropped because there was only one objection (causes implicit call to `global_stop_request()`), which causes the end of the Active Stage and start of Stop-Interrupt Stage).
  - No active `stop` task (starts running at time 500ns):
- `run()` phase finishes at time 500ns (end of both Stop-Interrupt Stage and `run()` phase).

```
class test9 extends uvm_test;
  `uvm_component_utils(test9)
  env e;

  function new (string name="test9", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #500ns;
    uvm_test_done.drop_objection();
  endtask
endclass
```

Example 26 - `test9` with `raise_objection()/drop_objection()` to terminate the test



The printed messages are shown in Figure 20 (UVM Report summary not shown). The first printed message came from the `run_test()` command itself, the next printed message came 500ns later and is the **TEST\_DONE** message, which shows that the dropping of all objections has forced an implicit call to the `global_stop_request()` command.

```
UVM_INFO @ 0: reporter [RNTST] Running test test9...
UVM_INFO @ 500: uvm_test_done [TEST_DONE]
    All end_of_test objections have been dropped.
    Calling global_stop_request()
```

Figure 20 - test9 UVM report output after all objections were dropped

The `test9` example shows that after all objections have been dropped that there is an implicit call to the `global_stop_request()` command, which will terminate the Active Stage of the `run()` phase.

Later tests will show that any active Objections-Raised threads will take control of when the Active Stage completes.

## 6.9 Raising And Dropping Multiple Objections Example

The **test10** code in Example 27 demonstrates the raising and dropping of objections from different components. Example 27 includes the code for the **test10** test and a testbench environment component (**env**).

The **top** module used for this test is the same simple **top** module used with **test7** (shown in Example 23).

The **test10** code has issued the **uvm\_test\_done.raise\_objection()** / **uvm\_test\_done.drop\_objection()** commands in the **run** task. The environment **env** component has also issued the **uvm\_test\_done.raise\_objection()** / **uvm\_test\_done.drop\_objection()** commands in its local copy of the **run** task. Once an objection is raised, all objections must be dropped before an implicit **global\_stop\_request()** will force the Active Stage to complete. The **test10 run** task drops its objection at time 500ns, while the **env run** task drops its objection at 700ns. The last dropped objection happens at time 700ns so that is when the Active Stage finishes. It is not necessarily recommended to override the **run** task in the environment component, but since **uvm\_env** is derived from the **uvm\_component** class, it is possible to add a **run** task to the environment.

When **test10** runs, the following sequence of actions will be executed:

- **test10** executes:
- **run()** phase starts at time 0ns (start of Active Stage).
  - **test10 run** task (starts running at time 0ns)
    - Call the **uvm\_test\_done.raise\_objection()** command.
    - Delays for 500ns.
    - Calls the **uvm\_test\_done.drop\_objection()** command at 500ns.
  - **env run** task (starts running at time 0ns)
    - Call the **uvm\_test\_done.raise\_objection()** command.
    - Delays for 700ns.
    - Calls the **uvm\_test\_done.drop\_objection()** command at 700ns.
    - All objections have now been dropped (causes implicit call to **global\_stop\_request()**), which causes the end of the Active Stage and start of Stop-Interrupt Stage).
  - No active **stop** task (starts running at time 700ns):
- **run()** phase finishes at time 700ns (end of both Stop-Interrupt Stage and **run()** phase).

```

class env extends uvm_env;
  `uvm_component_utils(env)

  function new (string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #700ns;
    uvm_test_done.drop_objection();
  endtask
endclass

class test10 extends uvm_test;
  `uvm_component_utils(test10)
  env e;

  function new (string name="test10", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #500ns;
    uvm_test_done.drop_objection();
  endtask
endclass

```

Example 27 - test10 & env with raise\_objection()/drop\_objection() - last drop ends test

The printed messages are shown in Figure 21 (UVM Report summary not shown). The first printed message came from the `run_test()` command itself, the next printed message came 700ns later and is the **TEST\_DONE** message, which shows that the dropping of all objections has forced an implicit call to the `global_stop_request()` command.

<pre> UVM_INFO @ 0: reporter [RNTST] Running test test10... UVM_INFO @ 700: uvm_test_done [TEST_DONE]     All end_of_test objections have been dropped.     Calling global_stop_request() </pre>
--

Figure 21 - test10 UVM report output after all objections were dropped

The `test10` example shows that after all objections have been dropped that there is an implicit call to the `global_stop_request()` command, which will terminate the Active Stage of the `run()` phase.

## 6.10 Multiple Objections And forever-Loop Example

The **test11** code in Example 29 demonstrates the raising and dropping of objections from different components and how they interact with another component that is executing **forever**-loop code a another **run** task. Example 28 includes the code for a testbench driver component (**tb\_driver**) and Example 29 includes the code for the **test11** test and environment component (**env**).

The **top** module used for this test is the same simple **top** module used with **test7** (shown in Example 23).

The **test11** code has issued the **uvm\_test\_done.raise\_objection()** / **uvm\_test\_done.drop\_objection()** commands in the **run** task. The environment **env** component has also issued the **uvm\_test\_done.raise\_objection()** / **uvm\_test\_done.drop\_objection()** commands in its local copy of the **run** task. The testbench driver **tb\_driver** has a **run** task that is executing a **forever** loop, but has no objections. Once an objection is raised, all objections must be dropped before an implicit **global\_stop\_request()** will force the Active Stage to complete. The **test11 run** task drops its objection at time 500ns, while the **env run** task drops its objection at 811ns. Dropping of all objections has forced an implicit call to the **global\_stop\_request()** command at 811ns, which does not require the **tb\_driver run** task to complete, so even though the **tb\_driver run** task may continue to run, the Active Stage completes with the last dropped objection and the Stop-Interrupt stage can now start. In **test11**, there are no enabled **stop** tasks, so the **run()** phase finishes right after the Active Stage finishes.

When **test11** runs, the following sequence of actions will be executed:

- **test11** executes:
- **run()** phase starts at time 0ns (start of Active Stage).
  - **test11 run** task (starts running at time 0ns)
    - Call the **uvm\_test\_done.raise\_objection()** command.
    - Delays for 500ns.
    - Calls the **uvm\_test\_done.drop\_objection()** command at 500ns.
  - **tb\_driver run** task (starts running at time 0ns)
    - **forever**-loop cycles every 120ns and prints a repeating message. The **forever**-loop keeps running until the **run()** phase finishes.
  - **env run** task (starts running at time 0ns)
    - Call the **uvm\_test\_done.raise\_objection()** command.
    - Delays for 811ns.
    - Calls the **uvm\_test\_done.drop\_objection()** command at 811ns.
    - All objections have now been dropped (causes implicit call to **global\_stop\_request()**), which causes the end of the Active Stage and start of Stop-Interrupt Stage).
  - No active **stop** task (starts running at time 811ns):
- **run()** phase finishes at time 700ns (end of both Stop-Interrupt Stage and **run()** phase).

```
class tb_driver extends uvm_driver;
  `uvm_component_utils(tb_driver)

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
  endfunction

  task run();
    forever begin
      #120 uvm_report_info("tb_driver", "driver run() loop");
    end
  endtask
endclass
```

Example 28 - test11 - tb\_driver with forever loop (loop will be cancelled when all objections are dropped)

```

class env extends uvm_env;
  `uvm_component_utils(env)
  tb_driver drv;

  function new (string name, uvm_component parent=null);
    super.new(name, parent);
    drv = tb_driver::type_id::create("drv", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #811ns;
    uvm_test_done.drop_objection();
  endtask
endclass

class test11 extends uvm_test;
  `uvm_component_utils(test11)
  env e;

  function new (string name="test11", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #500ns;
    uvm_test_done.drop_objection();
  endtask
endclass

```

Example 29 - test11 - env and test11 raise and drop objections

The printed messages are shown in Figure 22 (UVM Report summary not shown). The first printed message came from the `run_test()` command itself. The next six printed messages came at 120ns intervals from the `tb_driver` component. The next printed message came 811ns after the start of the Active Stage and is the `TEST_DONE` message, which shows that the dropping of all objections has forced an implicit call to the `global_stop_request()` command, even though the `forever` loop continued to run.

```

UVM_INFO @ 0: reporter [RNTST] Running test test11...
UVM_INFO @ 120: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 240: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 360: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 480: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 600: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 720: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 811: uvm_test_done [TEST_DONE]
      All end_of_test objections have been dropped.
      Calling global_stop_request()

```

Figure 22 - test11 UVM report output after all objections were dropped

The `test11` example shows that after all objections have been dropped that there is an implicit call to the `global_stop_request()` command, which will terminate the Active Stage of the `run()` phase, even if there was a `forever` loop running in another `run` task.

## 6.11 Multiple Objections Then Stop-Interrupt Activity Example

The **test12** code in Example 30 demonstrates the raising and dropping of objections from different components and how they interact with another component that has another **run** task that is executing a **forever**-loop and an enabled **stop** task in that same component. Example 30 includes the code for the **test12** test, a testbench driver component (**tb\_driver**) and environment component (**env**).

The **top** module used for this test is the same simple **top** module used with **test7** (shown in Example 23).

The **test12** code has issued the **uvm\_test\_done.raise\_objection()** / **uvm\_test\_done.drop\_objection()** commands in the **run** task. The environment **env** component has also issued the **uvm\_test\_done.raise\_objection()** / **uvm\_test\_done.drop\_objection()** commands in its local copy of the **run** task. The testbench driver **tb\_driver** has a **run** task that is executing a **forever** loop, but has no objections. The same **tb\_driver** also has an enabled **stop** task that runs another 1,000ns before returning. Once an objection is raised, all objections must be dropped before an implicit **global\_stop\_request()** will force the Active Stage to complete. The **test12 run** task drops its objection at time 500ns, while the **env run** task drops its objection at 811ns. Dropping of all objections has forced an implicit call to the **global\_stop\_request()** command at 811ns, which does not require the **tb\_driver run** task to complete, so even though the **tb\_driver run** task continues to run into the Stop-Interrupt Stage, the Active Stage completes with the last dropped objection and the Stop-Interrupt stage can now start. In **test12**, there is an enabled **stop** task, so the **run()** phase does not finish until after the **tb\_driver stop** task has finished. Throughout the Stop-Interrupt Stage, the **forever** loop continues to run and print its periodic repeating message.

When **test12** runs, the following sequence of actions will be executed:

- **test12** executes:
- **run()** phase starts at time 0ns (start of Active Stage).
  - **test12 run** task (starts running at time 0ns)
    - Call the **uvm\_test\_done.raise\_objection()** command.
    - Delays for 500ns.
    - Calls the **uvm\_test\_done.drop\_objection()** command at 500ns.
  - **tb\_driver run** task (starts running at time 0ns)
    - **forever**-loop cycles every 120ns and prints a repeating message. The **forever**-loop keeps running throughout the Stop-Interrupt Stage.
    - **stop** task (starts running at time 811ns).
    - print a "start of stop task " message
    - Delays for 1,000ns.
    - print an "end of stop task " message at 1,811ns.
  - **env run** task (starts running at time 0ns)
    - Call the **uvm\_test\_done.raise\_objection()** command.
    - Delays for 811ns.
    - Calls the **uvm\_test\_done.drop\_objection()** command at 811ns.
    - All objections have now been dropped (causes implicit call to **global\_stop\_request()**), which causes the end of the Active Stage and start of Stop-Interrupt Stage).
  - **tb\_driver stop** task (finishes at 1,811ns):
- **run()** phase finishes at time 1,811ns (end of both Stop-Interrupt Stage and **run()** phase).

```

class tb_driver extends uvm_driver;
  `uvm_component_utils(tb_driver)

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
  endfunction

  task run();
    enable_stop_interrupt = '1;
    forever begin
      #120 uvm_report_info("tb_driver", "driver run() loop");
    end
  endtask

  task stop(string ph_name);
    uvm_report_info("tb_driver", "start of stop task");
    #1000;
    uvm_report_info("tb_driver", "end   of stop task");
  endtask
endclass

class env extends uvm_env;
  `uvm_component_utils(env)
  tb_driver drv;

  function new (string name, uvm_component parent=null);
    super.new(name, parent);
    drv = tb_driver::type_id::create("drv", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #811ns;
    uvm_test_done.drop_objection();
  endtask
endclass

class test12 extends uvm_test;
  `uvm_component_utils(test12)
  env e;

  function new (string name="test12", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #500ns;
    uvm_test_done.drop_objection();
  endtask
endclass

```

Example 30 - test12 - tb\_driver with cancelled forever loop but still executes stop task

The printed messages are shown in Figure 23 (UVM Report summary not shown). The first printed message came from the `run_test()` command itself. The next six printed messages came at 120ns intervals from the `tb_driver` component. The next printed message came 811ns after the start of the Active Stage and is the `TEST_DONE` message, which shows that the dropping of all objections has forced an implicit call to the `global_stop_request()` command, even though the `forever` loop continued to run. The next message came from the `tb_driver stop` task while the `tb_driver run` task continued to execute the `forever` loop. The next nine printed messages continued to come at 120ns intervals from the `tb_driver` component, while the last printed message came from the `tb_driver stop` task just before the Stop-Interrupt Stage finished.

```
UVM_INFO @ 0: reporter [RNTST] Running test test12...
UVM_INFO @ 120: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 240: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 360: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 480: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 600: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 720: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 811: uvm_test_done [TEST_DONE]
    All end of test objections have been dropped.
    Calling global_stop_request()
UVM_INFO @ 811: uvm_test_top.e.drv [tb_driver] start of stop task
UVM_INFO @ 840: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 960: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1080: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1200: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1320: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1440: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1560: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1680: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1800: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 1811: uvm_test_top.e.drv [tb_driver] end of stop task
```

Figure 23 - test12 UVM report output after all objections were dropped and stop task completed

The `test12` example shows that after all objections have been dropped that there is an implicit call to the `global_stop_request()` command, which will terminate the Active Stage of the `run()` phase, even if there was a `forever` loop running in another `run` task. It also shows that even though `test12` entered the Stop-Interrupt Stage for the next 1,000ns, that the `run` task `forever` loop continued to run throughout the Stop-Interrupt stage. It can be seen that not all `run` tasks have to finish before the Stop-Interrupt Stage activity can commence.



## 6.12 Dropped Objection And Early `global_stop_request()` Example

The `test13` code in Example 32 demonstrates that the raising and dropping of even a single objection will disable an Active Stage call to `global_stop_request()` from a `run` task in the test. Example 31 includes the code for a testbench driver component (`tb_driver`) and Example 32 includes the code for the `test13` test and environment component (`env`).

The `top` module used for this test is the same simple `top` module used with `test7` (shown in Example 23).

The environment `env` component has issued the `uvm_test_done.raise_objection()` / `uvm_test_done.drop_objection()` commands in its local copy of the `run` task. The `test13` code has issued a `global_stop_request()` command in its `run` task before the `env` issued the drop objection command. The testbench driver `tb_driver` has a `run` task that is executing a `forever` loop, but has no objections. Once an objection is raised, all objections must be dropped before an implicit `global_stop_request()` will force the Active Stage to complete. The `test13` call to `global_stop_request()` happened while there was an active raised objection, so the `test13 global_stop_request()` command was effectively ignored until all objections were dropped. In `test13`, there are no enabled `stop` tasks, so the `run()` phase finishes right after the Active Stage finishes.

When `test13` runs, the following sequence of actions will be executed:

- `test13` executes:
- `run()` phase starts at time 0ns (start of Active Stage).
  - `test13 run` task (starts running at time 0ns)
    - Delays for 100ns.
    - Calls the `global_stop_request()` command at 100ns (ignored)
  - `tb_driver run` task (starts running at time 0ns)
    - `forever`-loop cycles every 120ns and prints a repeating message. The `forever`-loop keeps running throughout the Active Stage.
  - `env run` task (starts running at time 0ns)
    - Call the `uvm_test_done.raise_objection()` command.
    - Delays for 81ns.
    - Calls the `uvm_test_done.drop_objection()` command at 81ns.
    - All objections have now been dropped (causes implicit call to `global_stop_request()`), which causes the end of the Active Stage and start of Stop-Interrupt Stage).
  - No active `stop` task (starts running at time 81ns):
- `run()` phase finishes at time 81ns (end of both Stop-Interrupt Stage and `run()` phase).

```
class tb_driver extends uvm_driver;
  `uvm_component_utils(tb_driver)

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
  endfunction

  task run();
    forever begin
      #120 uvm_report_info("tb_driver", "driver run() loop");
    end
  endtask
endclass
```

Example 31 - `test13` - `tb_driver` with forever loop (loop will be cancelled when all objections are dropped)

```

class env extends uvm_env;
  `uvm_component_utils(env)
  tb_driver drv;

  function new (string name, uvm_component parent=null);
    super.new(name, parent);
    drv = tb_driver::type_id::create("drv", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #811ns;
    uvm_test_done.drop_objection();
  endtask
endclass

class test13 extends uvm_test;
  `uvm_component_utils(test13)
  env e;

  function new (string name="test13", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    #100ns;
    global_stop_request();
  endtask
endclass

```

Example 32 - test13 - global\_stop\_request() ignored until env drops objections

The printed messages are shown in Figure 24 (UVM Report summary not shown). The first printed message came from the `run_test()` command itself. The next six printed messages came at 120ns intervals from the `tb_driver` component. The next printed message came 811ns after the start of the Active Stage and is the **TEST\_DONE** message, which shows that the dropping of all objections has forced an implicit call to the `global_stop_request()` command, even though the **forever** loop continued to run. NOTE: there was a call to `global_stop_request()` at 100ns, but the command was ignored until all objections were dropped. The final message makes reference to the "Previous call to `global_stop_request()`," which is "now (being) honored."

```

UVM_INFO @ 0: reporter [RNTST] Running test test13...
UVM_INFO @ 120: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 240: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 360: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 480: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 600: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 720: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 811: uvm_test_done [TEST_DONE]
    All end_of_test objections have been dropped.
    Previous call to global_stop_request() will now be honored.

```

Figure 24 - test13 UVM report output after all objections were dropped

The `test13` example shows that explicit calls to `global_stop_request()` are ignored if there are any raised objections. Only after all objections have been dropped is there an implicit call to the `global_stop_request()` command, which will terminate the Active Stage of the `run()` phase.

### 6.13 Dropped Objection And Late global\_stop\_request() Example

The **test14** code in Example 34 demonstrates that the raising and dropping of even a single objection will disable any future call to **global\_stop\_request()** from a **run** task in the test. Example 33 includes the code for a testbench driver component (**tb\_driver**) and Example 34 includes the code for the **test14** test and environment component (**env**).

The **top** module used for this test is the same simple **top** module used with **test7** (shown in Example 23).

The environment **env** component has issued the **uvm\_test\_done.raise\_objection()** / **uvm\_test\_done.drop\_objection()** commands in its local copy of the **run** task. The **test14** code has issued a **global\_stop\_request()** command in its **run** task after the **env** issued the drop objection command. The testbench driver **tb\_driver** has a **run** task that is executing a **forever** loop, but has no objections. Once an objection is raised, all objections must be dropped before an implicit **global\_stop\_request()** will force the Active Stage to complete. The **test14** call to **global\_stop\_request()** happened after there was an active raised objection, so the **test14 global\_stop\_request()** command was never executed. In **test14**, there are no enabled **stop** tasks, so the **run()** phase finishes right after the Active Stage finishes.

When **test14** runs, the following sequence of actions will be executed:

- **test14** executes:
- **run()** phase starts at time 0ns (start of Active Stage).
  - **test14 run** task (starts running at time 0ns)
    - Delays for 1,000ns.
    - Calls the **global\_stop\_request()** command at 1,000ns (too late - **run()** phase has already completed)
  - **tb\_driver run** task (starts running at time 0ns)
    - **forever**-loop cycles every 120ns and prints a repeating message. The **forever**-loop keeps running throughout the Active Stage.
  - **env run** task (starts running at time 0ns)
    - Call the **uvm\_test\_done.raise\_objection()** command.
    - Delays for 811ns.
    - Calls the **uvm\_test\_done.drop\_objection()** command at 811ns.
    - All objections have now been dropped (causes implicit call to **global\_stop\_request()**), which causes the end of the Active Stage and start of Stop-Interrupt Stage).
  - No active **stop** task (starts running at time 811ns):
- **run()** phase finishes at time 811ns (end of both Stop-Interrupt Stage and **run()** phase).

```
class tb_driver extends uvm_driver;
  `uvm_component_utils(tb_driver)

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
  endfunction

  task run();
    forever begin
      #120 uvm_report_info("tb_driver", "driver run() loop");
    end
  endtask
endclass
```

Example 33 - test14 - tb\_driver with forever loop (loop will be cancelled when all objections are dropped)

```

class env extends uvm_env;
  `uvm_component_utils(env)
  tb_driver drv;

  function new (string name, uvm_component parent=null);
    super.new(name, parent);
    drv = tb_driver::type_id::create("drv", this);
  endfunction

  task run;
    uvm_test_done.raise_objection();
    #811ns;
    uvm_test_done.drop_objection();
  endtask
endclass

class test14 extends uvm_test;
  `uvm_component_utils(test14)
  env e;

  function new (string name="test14", uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build;
    super.build();
    e = env::type_id::create("e", this);
  endfunction

  task run;
    #1000ns;
    global_stop_request();
  endtask
endclass

```

Example 34 - test14 - global\_stop\_request() comes too late after all objections dropped and is ignored

The printed messages are shown in Figure 25 (UVM Report summary not shown). The first printed message came from the `run_test()` command itself. The next six printed messages came at 120ns intervals from the `tb_driver` component. The next printed message came 811ns after the start of the Active Stage and is the **TEST\_DONE** message, which shows that the dropping of all objections has forced an implicit call to the `global_stop_request()` command, even though the **forever** loop continued to run. NOTE: there was a call to `global_stop_request()` at 1,000ns, but the command came after the `run()` phase had finished.

```

UVM_INFO @ 0: reporter [RNTST] Running test test14...
UVM_INFO @ 120: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 240: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 360: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 480: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 600: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 720: uvm_test_top.e.drv [tb_driver] driver run() loop
UVM_INFO @ 811: uvm_test_done [TEST_DONE]
      All end_of_test objections have been dropped.
      Calling global_stop_request()

```

Figure 25 - test14 UVM report output after all objections were dropped

The `test14` example shows that after all objections have been dropped there is an implicit call to the `global_stop_request()` command, which will terminate the Active Stage of the `run()` phase.

## 7. SUMMARY & GUIDELINES

The UVM (and OVM) `run()` phase is composed of two stages: the Active Stage that runs unconditionally, and the Stop-Interrupt Stage that executes conditionally if any `enable_stop_interrupt` bits were set during the Active Stage.

There are three types of threads that can be started at the beginning of the Active Stage when the `run_test()` command is executed. These threads are the *Non-Stopping* threads, *Stop-Request* threads and *Objections-Raised* threads.

If there are any Objections-Raised threads, the `global_stop_request()` command is largely ignored. For this reason, the `global_stop_request()` command is probably not the best way to terminate the Active Stage of the `run()` phase. All it takes is a single Objections-Raised thread to invalidate all of the `global_stop_request()` commands in all of the Stop-Request threads. The Stop-Request threads effectively become Non-Stopping threads if there are any active Objection-Raised threads.

UVM testbench development follows a different coding pattern than the typical Verilog testbench style. Understanding the differences will help the UVM verification engineer to successfully build efficient UVM tests that gracefully terminate without timing out. To help build correct UVM testbenches, we offer the following guidelines to successfully setup UVM tests that gracefully terminate.

**Guideline:** Do not use a `$finish` command in a UVM testbench. If executed, the `$finish` command will abort the UVM `run()` phase and never execute the post-`run()` phases.

**Guideline:** Do not place any `global_stop_request()` or `$finish` command in the test file after the `run_test()` command. The stop or finish command will never execute and the simulation will either hang or timeout.

**Guideline:** In general, avoid using the `global_stop_request()` command. This command does not work if any active objections have been raised and its apparent failure to terminate at the expect time is a point of *global\_stop\_confusion!*

**Guideline:** Only use the `global_stop_request()` command in the simplest of tests where you have control over all of the verification components.

**Guideline:** Use the `uvm_test_done.raise_objection()` / `uvm_test_done.drop_objection()` commands in all test and component `run` tasks to control when that test or component has finished its testing activity.

**Guideline:** When using the stop interrupt capabilities, set the `enable_stop_interrupt` bit at the beginning of the `run` task. Setting this bit will allow the `run()` phase to execute the Stop-Interrupt Stage.

**Guideline:** Any test or component that sets the `enable_stop_interrupt` bit should also include a stop task to describe the testing activity that should be executed during the Stop-Interrupt Stage or the `run()` phase.

UVM simulations rarely hang, but if the tests are improperly terminated, the simulations will run for a very long time and appear to be hung. A good test will never need to set either the `phase_timeout` or `stop_timeout` limits. The test will properly execute and gracefully terminate on its own. If the simulation run seems to be hanging, then the timeout limits can be added to check for a never-ending test.

**Guideline:** Do not call the `set_global_timeout()` command in the `run` task (it is too late to become active)

**Guideline:** A good place to use the `set_global_timeout()` and `set_global_stop_timeout()` commands is in an `initial` block in the `top`-module before calling the `run_test()` command.

Following these guidelines will help avoid frustration related to improper UVM test termination.

## 8. ACKNOWLEDGMENTS

Our thanks to our colleague Kelly Larson for sharing useful comments and feedback during the development of this paper. Also thanks to our colleague Mike Horn for sharing important information related to the use of the `run_test()` command.

## 9. REFERENCES

- [1] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2009. <http://standards.ieee.org/findstds/standard/1800-2009.html>
- [2] Mark Glasser, "Open Verification Methodology Cookbook", Springer, [www.springeronline.com](http://www.springeronline.com), 1st Edition., 2009. ISBN: 978-1-4419-0967-1  
Free PDF Version at: <http://verification-academy.mentor.com/content/open-verification-methodology-advanced-ovm-uvm-module>
- [3] Mentor Graphics Corp. 10 Dec 2010. *EOT/Guide*. [http://uvm.mentor.com/mc/EOT/ovm\\_test\\_done](http://uvm.mentor.com/mc/EOT/ovm_test_done)
- [4] Mentor Graphics Corp. 28 Feb 2011. *UVM/OVM Methodology Cookbook*. <http://uvm.mentor.com/uvm/EOT/Guide>
- [5] Mike Horn - personal communication.
- [6] OVM Class Reference, Version 2.1.1, March 2010. In OVM source code: `ovm-2.1.1/OVM_Reference.pdf`

- [7] OVM 2.1.1 kit - includes OVM base class libraries - Free downloads - [www.ovmworld.org](http://www.ovmworld.org) (choose Download).
- [8] `ovm-2.1.1/src/base/ovm_root.svh` - comments in the source code file.
- [9] Universal Verification Methodology (UVM) 1.0 EA Class Reference, May 2010, Accellera, Napa, CA. In UVM source code: `uvm1.0ea/uvm_reference_guide_1.0_ea.pdf`
- [10] UVM 1.0EA (Early Adopter) kit - includes UVM base class libraries - Free downloads - [www.uvmworld.org](http://www.uvmworld.org) (choose Download).
- [11] `uvm1.0ea/src/base/uvm_root.svh` - comments in the source code file.

## 10. AUTHOR & CONTACT INFORMATION

**Cliff Cummings**, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 29 years of ASIC, FPGA and system design experience and 19 years of SystemVerilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past eight years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the [www.sunburst-design.com](http://www.sunburst-design.com) web site.

Email address: [cliffc@sunburst-design.com](mailto:cliffc@sunburst-design.com)

**Tom Fitzpatrick**, Verification Technologist and Editor of Verification Horizons at Mentor Graphics. Design and verification expert using SystemVerilog; developing, educating and writing about industry standards.

Email address: [top\\_fitzpatrick@mentor.com](mailto:top_fitzpatrick@mentor.com)

An updated version of this paper can be downloaded from the web sites:

**[www.sunburst-design.com/papers](http://www.sunburst-design.com/papers)**