# A VMM based generic interrupt handling mechanism

Ning Guo, Rich Musacchio
Steve D’onofrio, Ambar Sarkar

Paradigm Works

ning.guo@paradigm-works.com
rich.musacchio@paradigm-works.com
stephen.donofrio@paradigm-works.com
ambar.sarkar@paradigm-works.com

**ABSTRACT**

Interrupt monitoring and checking is an important task in a verification process and it often reveals design bugs. However, interrupts are often tested as special case scenarios because they are generally associated with some deviation from the normal operation of the device under test.
Since both design and verification tasks usually focus on the normal functionalities first, special scenario testing may start very late in the verification process. If a verification environment is built up without considering various unusual conditions, adding interrupt testing capabilities late in the verification development cycle can significantly disrupt the existing verification environment.
This paper presents a generic interrupt handling mechanism which can be employed early in the verification development process and hence eliminate the risk of retrofitting it in later.
Common issues of interrupt monitoring and servicing process are also addressed and some strategies to handle those issues are proposed.

# Table of Contents


1.0 Introduction ........ 4
2.0 Environment Architecture Overview ........ 4
3.0 Generate Interrupt Conditions ........ 6
4.0 Interrupt Status and Scoreboard ........ 8
*4.1 Base Interrupt Status class* ........ 8
*4.2 Interrupt Scoreboarding* ........ 9
5.0 Interrupt Handling Transactor ........ 11
*5.1 Base interrupt handler* ........ 12
*5.2 Interrupt Monitoring Process* ........ 13
*5.3 Interrupt Status Accessing Process* ........ 14
*5.4 Check interrupt status* ........ 17
*5.5 Complete Interrupt Servicing Process* ........ 18
*5.6 End-of-test interrupt status checking* ........ 19
6.0 Fast Posting and Slow Checking ........ 20
*6.1 Interrupt from multiple sources* ........ 20
*6.2 Multiple interrupts from same source* ........ 21
*6.3 Single interrupt status from multiple sources* ........ 21
7.0 Other considerations for interrupt verification ........ 21
*7.1 At least two rounds of interrupt service* ........ 21
*7.2 Cooperate with other register accessing processes in the verification environment* ........ 21
8.0 Conclusion ........ 22
9.0 References ........ 23

# Table of Figures

Figure 1 VMM verification environment with layered interface transactor ....... 5
Figure 2 Interrupt verification components in relation to layered interface transactor ....... 6
Figure 3 Use transaction constraint to generate interrupt condition ....... 7
Figure 4 Use generator callback extension to produce interrupt condition ....... 7
Figure 5 Base status class ....... 8
Figure 6 Single Interrupt and Slow Posting – One scoreboard entry queue ....... 9
Figure 7 Multiple Interrupts and Fast Posting – Multiple scoreboard queues ....... 10
Figure 8 Example of scoreboarding expected interrupt status ....... 11
Figure 9 Interrupt Handler Class Diagram ....... 12
Figure 10 Base Interrupt Handler ....... 12
Figure 11 wait_for_interrupt_t() implementation example ....... 14
Figure 12 Example read_status_t() and Register Model ....... 16
Figure 13 Pseudo codes of accessing multiple status registers or status associative registers ....... 17
Figure 14 Example check_status() code ....... 18
Figure 15 Generic main() process of base interrupt handler ....... 19
Figure 16 Example process_interrupt_t() ....... 19
Figure 17 Check interrupt status in cleanup phase ....... 20
Figure 18 Manager register access order through Semaphore in register factory class ....... 22

## 1.0 Introduction

Interrupt features are incorporated by almost all designs. Therefore, interrupt monitoring and checking is very important task in the verification process and it often reveals many design bugs. A commonly employed design feature is that when something went wrong or when some information needs to be passed to software, an interrupt is triggered. Meanwhile the causes of the interrupt are stored in status registers. This will allow software to read the register value and find out the reason of the interrupt.

In order to verify such a feature, verification environment needs to 1) generate interrupt triggering conditions; 2) Monitor for the interrupt indication; 3) Mimic software to read from the registers which stored interrupt specific information and 4) compare against the expected interrupt status. In addition, interrupts may be disabled or masked and we need to verify that the appropriate statuses are set but this condition does not assert an interrupt. However, interrupts are often tested as special case scenarios because they are generally associated with some deviation from the normal operation of the device under test.

Since both design and verification usually focus on the normal functionalities, special scenario testing may start very late in the verification process. If a verification environment is built up without considering various exceptional conditions, adding interrupt testing capabilities late in the verification development cycle could have unexpected interference with the existing verification environment.

This paper presents a VMM based generic interrupt handling mechanism which can be employed early in the verification development process and hence eliminate the risk of retrofitting it in later. The mechanism takes into account all four steps stated above for verifying the interrupt features of a design. Section 2 presents the VMM based environment architecture from which the proposed interrupt handling mechanism can be applied; Sections 3 and 4 focuses on the transmit side of the interrupt verification process, i.e., interrupt stimulus generation, interrupt status representation and scoreboarding; Section 5 discusses the receive side of the interrupt handling scheme, which is implemented through generic interrupt handler class; section 6 talks about the asymmetric posting and checking feature of the interrupt testing; section 7 covers a few additional concerns for using the proposed mechanism; section 8 concludes the article.

## 2.0 Environment Architecture Overview

In a VMM based verification environment, an interface transactor, which verifies interface specific protocol, is built with VMM layered architecture. Each transactor has transmit side that is responsible for generating stimulus, driving the stimulus to the DUT following the appropriate protocol; and receive side which samples DUT's responses, reassembles bus activities to a higher level of abstraction data structure and passes to upper layer. If the transactor needs to be self-checking, a scoreboard will be included in the structure. The transmit side of the transactor posts expected DUT responses to the scoreboard; the receive side pulls out expected DUT responses from scoreboard and compares against the real DUT's responses. Figure 1 shows the block

diagram of a self-checking layered transactor. A complete verification environment consists of one or multiple layered transactors depending on how many interfaces a DUT has.

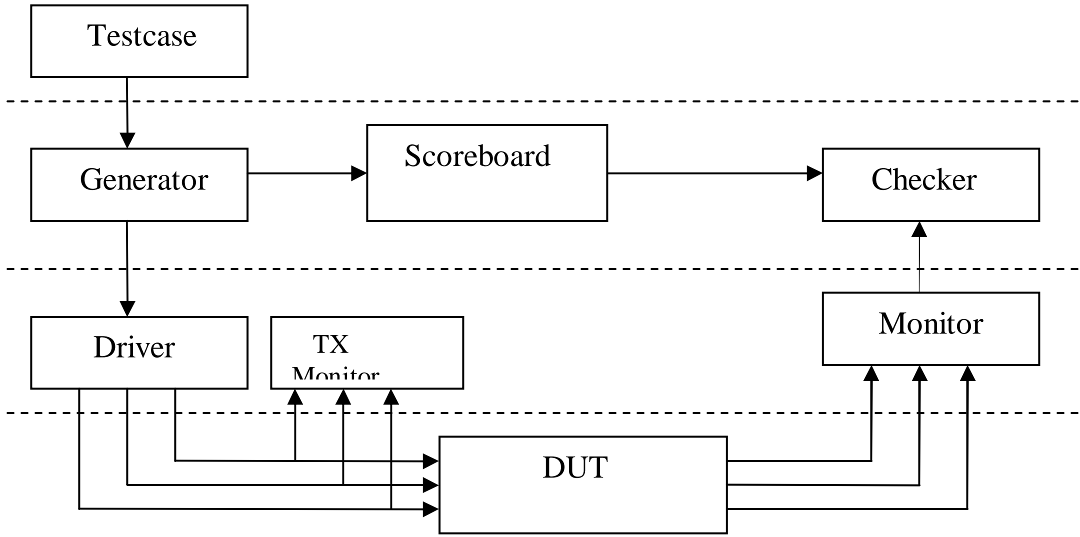


**Figure 1 VMM verification environment with layered interface transactor**

Interrupt verification can be viewed the same way as interface verification. It needs to have transmit side which is responsible for generating interrupt conditions, and receive side which detects the interrupt responses and conducts checking. However, unlike interface verification, interrupt verification does not need its own driver because all it does is to deviate from normal operation. Therefore, interrupt generation simply uses existing interface transactor and makes necessary modifications on the normal transactor flow using the hooks provided by the interface transactor. Details on interrupt generation will be discussed in Section 3.

In terms of the receive side of interrupt verification, normally there will be interrupt specific physical signals from DUT, thus an interrupt specific monitor is needed. An interrupt specific checker is also needed due to the uniqueness of interrupt processing. Section 4 will describe the receive side of interrupt handling mechanism in depth.

Figure 2 shows the same interface transactor structure as of Figure 1 but with interrupt verification components added to it. The shaded blocks represent components needed for interrupt verification. The scoreboard block is using slight different shading because it contains entries for both interrupt and non-interrupt components. Figure 2 also illustrates that when a verification environment has multiple interface transactors, there will have multiple generator or driver hooks for interrupt stimulus generation (i.e., one for each interface transactor), but only one copy of interrupt monitor and checker are needed. Section 4 will explain about this.

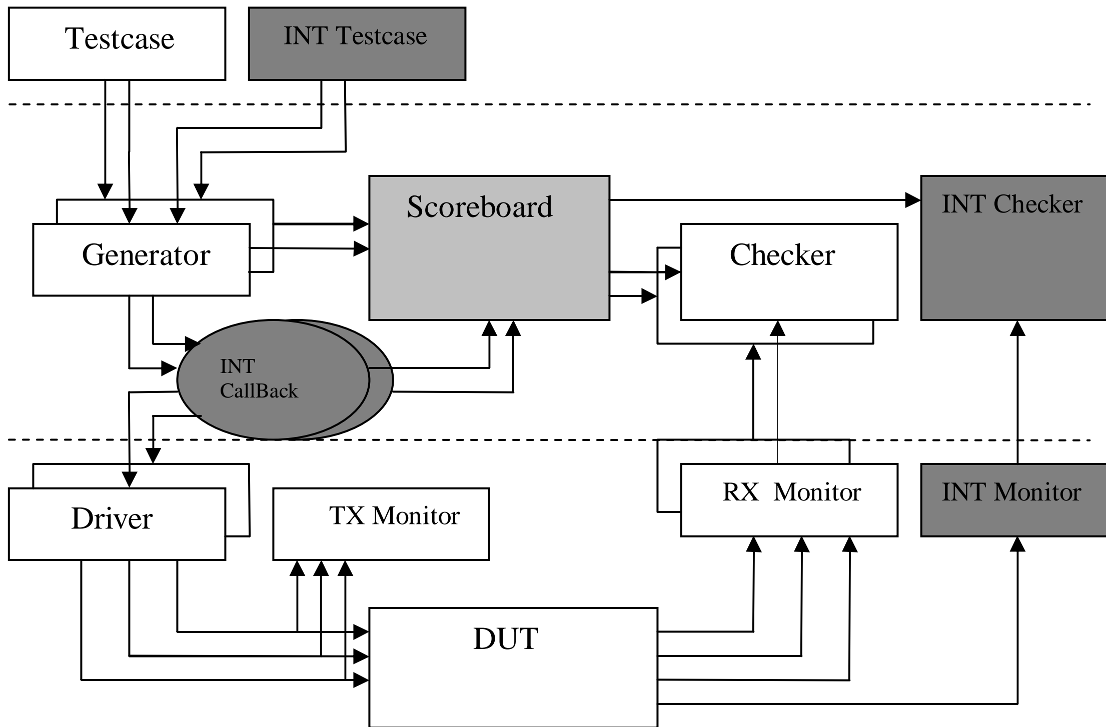


**Figure 2 Interrupt verification components in relation to layered interface transactor**

## 3.0 Generate Interrupt Conditions

Interrupt can be introduced through data variation or control variation. In VMM based constrained random verification environment, data variation can be generated by constraining transaction class property to be in a particular range, or by pre- or post- processing normally generated data; control variation can be done through constraining transactor's programmable parameters to a certain operating range, or through pre- or post- processing in callback methods.

If an interrupt condition is produced through constraining transaction object, one can create a derived transaction class which has the interrupt triggering constraint block turned on, and use this derived transaction class as the factory pattern of the generator. Such derived transaction class can be created in individual interrupt testcase (i.e., the INT testcase block in Figure 2), or it can be put in a DUT specific data class library file when many derived data classes need to be created and shared by multiple testcases. Similarly, if an interrupt condition can be activated by constraining programmable parameters of a transactor, one can create a derived parameter class for that transactor and constrain the parameters to the proper range. This derived transactor parameter class can also be implemented in interrupt testcase or as an independent file.

In the case that an interrupt condition could not be easily generated using constraints, transactor callback can be used to create processes which will modify the normally generated stimulus or inject exceptions.

Depending on the transactor setup and the transactions being processed, one of the above mentioned interrupt generation schemes or combinations of them can be employed in a verification environment.

Figure 3 and Figure 4 showed example codes of the two schemes.

```
// Normal transaction class
class default_trans extends vmm_data;
  rand bit[7:0] bytes[];
  constraint byte_size_constr { bytes.size() < 20; }
  constraint byte_value_constr { foreach(bytes[i]) bytes[i]<8'h80;}
endclass

// Derived transaction class for generating bad sized transaction
class error_trans extends default_trans;
  constraint size_err { bytes.size() == 30;}
endclass // error_trans
```

**Figure 3 Use transaction constraint to generate interrupt condition**

```
// Base callback class for "trans_gen" transactor class
class trans_gen_cb extends vmm_xactor_callbacks;
    virtual task post_gen(trans_gen gen, ref default_trans tr);
    endtask
endclass

// Transaction generator calss
class trans_gen extends vmm_xactor;
    task main();
       default_trans tr;
       if (!randomized_obj.randomize()) …
       $cast(tr, randomized_obj.copy());
       `vmm_callback(trans_gen_cb, post_gen(this, tr);
        ….
    endtask
endclass

// Derived callback class which corrupts data bytes
class error_trans_gen_cb extends trans_gen_cb;
     task post_gen(trans_gen gen, ref default_trans tr);
         foreach(tr.bytes[i]) // corrupt even bytes
             if (i%2==0) tr.bytes[i] = tr.bytes[i]+8'h80;
     endtask
```

**Figure 4 Use generator callback extension to produce interrupt condition**

## 4.0 Interrupt Status and Scoreboard

In order to make interrupt verification process self-checking, the interrupt handling mechanism uses scoreboard to store predicted interrupt responses. As shown in Figure 2, interrupt can use the same scoreboard as the one used for regular data scoreboarding. To achieve this, we need to define proper scoreboard entry to represent interrupt responses.

### *4.1 Base Interrupt Status class*

When an interrupt happened, in order for software to find out which interrupt source triggered the interrupt, hardware stores interrupt source information in interrupt status registers. Each register field corresponds to a unique interrupt source. Depending on the complexity of interrupt logic, a DUT could have single or multiple interrupt status registers.
Besides interrupt source information, some interrupts need additional registers to store other useful information to augment interrupt analyzing process. For example, if a memory access failed and triggered an interrupt, the memory address can be stored in a register so that software can read the register and find out which memory space is corrupted.

Regardless of what information is stored, the commonality is that these interrupt specific information are stored in registers. In verification environment, in order to support self-checking for interrupt testing, a data structure which carries the expected registered information of a particular interrupt source needs to be created and made accessible to the receive side of the environment so that it will be later on compared with the responses obtained from DUT.

There are many ways to represent registers in a verification environment. Register modeling scheme could be very different for different companies or different groups in the same company which makes it very difficult for reusing. In order to for better reusing, instead of building interrupt status class upon register representation, we declare a generic standalone base interrupt status class.  Figure 5 shows the base status class code.

```
class base_status extends vmm_data;
    stat_id_e          status_id;
    string             status_name;
    int                status_value;
    int                object_id;
    bit[63:0]          address[string];
    //   Function/Tasks
    …..
endclass // base_status
```

**Figure 5  Base status class**

In base_status class, the ***status_id*** property stores the identification of each status object. The ***status_id*** field can be declared as an enumerator type or an integer type. The advantage of using enumerator type is that it makes functional coverage collection easier because one can directly use ***status_id*** field as a cover point and the enumerator values are automatically the bins of the

cover point. The ***status_id*** property is needed when posting or pulling status entries to and from scoreboard. Section 4.2 will talk more on the scoreboarding process.
The ***status_value*** field stores the expected value of the status field identified by the ***status_id***.
The ***status_name*** field concatenates the status register name string and the status field name string. It can be used to derive the ***status_id*** of a status object. Section 5 will show an example usage of ***status_name***.
The ***object_id*** field indicates how many this same kind of status objects have been put on scoreboard.

The ***address*** field is the place holder for the case when interrupt is associated with a particular memory space, and DUT stores the corrupted memory space address. The ***address*** field is represented by string indexed associative array. The index is the name of the register which stores the address information. If no additional address registers are used in a design, the array is empty.

### *4.2 Interrupt Scoreboarding*

As we have mentioned in previous sections, in order to achieve self-checking, an expected interrupt status object needs to be formed and put on scoreboard whenever an interrupt triggering condition is enabled. For example, if interrupt stimulus is created by transmitting particularly constrained transaction objects, whenever such transaction is generated, an expected interrupt status object needs to be created and posted onto scoreboard if the interrupt is enabled; if an interrupt is triggered through callback method, expected interrupt status object must be built and posted on scoreboard from the same callback method.

If interrupts are triggered one at a time, and the interval between two consecutive interrupts are long enough so that servicing for the previous interrupt always completes before next interrupt is triggered, then all expected status objects can be posted to the same queue on the scoreboard as shown in Figure 6.

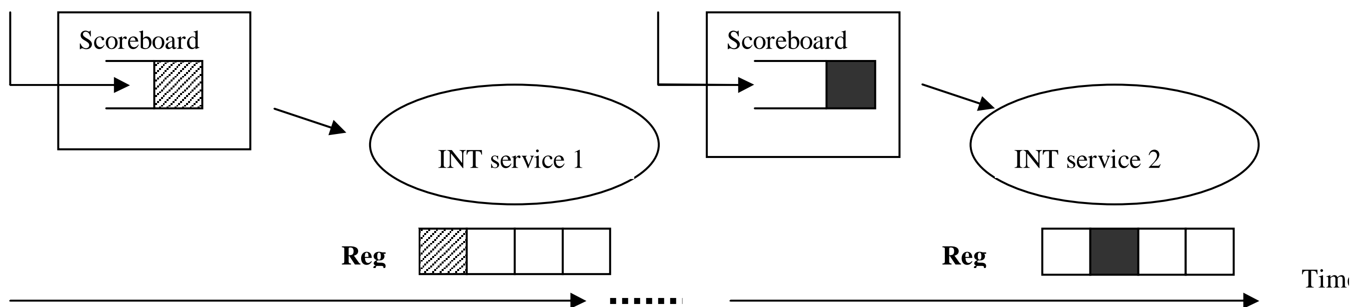


**Figure 6  Single Interrupt and Slow Posting – One scoreboard entry queue**

However, in reality, multiple interrupts could happen at the same time and the interval between two interrupts could be short. To handle these circumstances, we can create multiple queues to store expected statuses. The queues are distinguished using status register and field information because register and status field determines the unique interrupt response. In the base_status class of Figure 5, we have mentioned that the ***status_id*** field is to uniquely identify an interrupt status, so it can be used as scoreboard queue identification. Figure 7 shows the scoreboard with multiple status queues. Section 6 will discuss more about scoreboarding and checking for fast posting scenario.

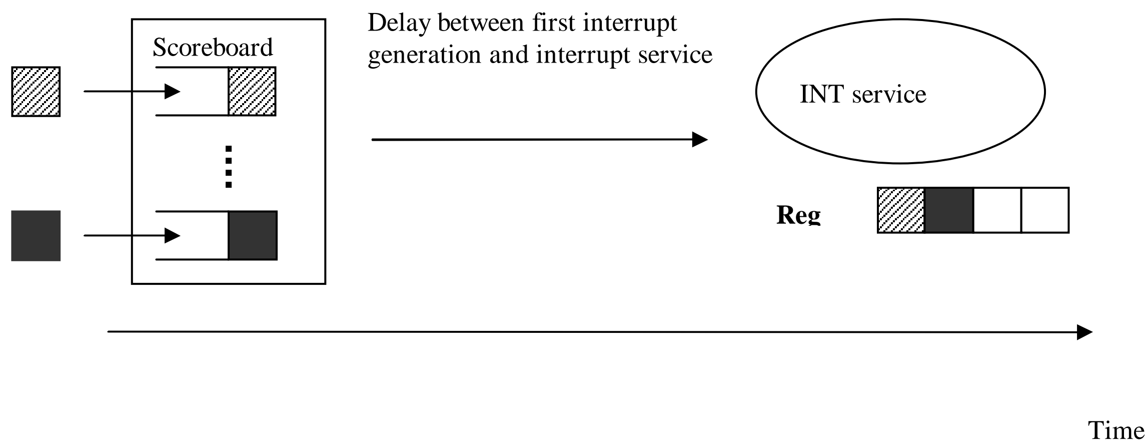


**Figure 7 Multiple Interrupts and Fast Posting – Multiple scoreboard queues**

Figure 8 presents example codes of posting an expected interrupt status onto a scoreboard.

```
// Using generator callback to introduce interrupt condition and scoreboard expects
class error_gen_cb extends trans_gen_cb;
  base_scoreboard   stat_sb;    // Handle to scoreboard

  function new(base_scoreboard stat_sb);
      If (stat_sb != null) this.stat_sb = stat_sb;
  endfunction

  virtual task post_gen (trans_gen gen, ref default_trans tr);
      base_status   exp_status;
       foreach(tr.bytes[i]) // corrupt even bytes
         if (i%2==0) begin
            tr.bytes[i] = tr.bytes[i]+8'h80;
            // Build expect status object
             exp_status = new;
             exp_status.reg_name = "INT_STAT";
             exp_status.status_value = 1;
             exp_status.status_id = e_INT_STAT_BYTE_ERR;
            // Built-in task of base_scoreboard: post(entry, flow_id)
            sb.post(exp_status, exp_status.status_id);
          end
      end
  endtask
  ….
endclass
```

**Figure 8 Example of scoreboarding expected interrupt status**

The above example assumes that the base_scoreboard class has a built-in *post()* task. The task puts the specified expected status object to the appropriate queue on the scoreboard. The queue is identified by the ***status_id*** of the expected status object.

# 5.0 Interrupt Handling Transactor

Section 4 discussed about generating interrupt stimulus and scoreboard expected interrupt status, which are the processes happening on the transmit side of interrupt testing. This section is going to focus on the receive side of the interrupt handling mechanism. As we have shown in Figure 2, the receive side of interrupt verification include monitor and checker. Monitor is responsible for determining when interrupt service needs to be activated, checker takes care of accessing status registers and comparing read back status values against expected status.

Unlike the monitor of layered interface transactor, the interrupt monitor does not need to reassemble bus activities to higher abstraction data structure, so it can directly communicate with checker without using a VMM channel. Therefore, the complete receiving processes can be implemented in one transactor class as shown in Figure 9.

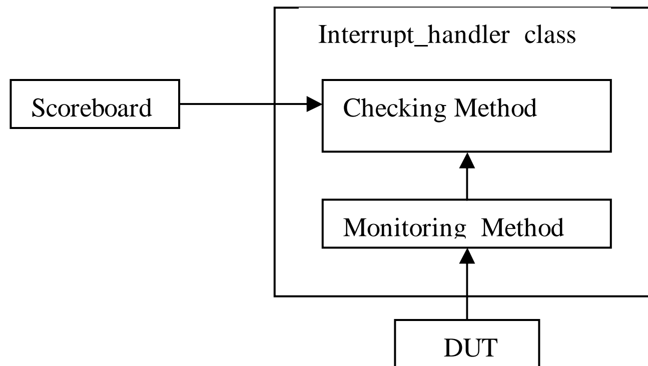


**Figure 9 Interrupt Handler Class Diagram**

Although the interrupt indications and the status register accessing procedures are specific for each design, the main flow of interrupt detecting and servicing is generic and hence can be built into a base interrupt handling transactor that can be reused.

### *5.1 Base interrupt handler*

The base interrupt handler extends from vmm_transactor. It comprises the following interrupt receiving processes, i.e., interrupt monitoring, status reading and status checking. Each process is implemented as a virtual task, i.e., wait_for_interrupt_t(), read_status_t() and check_status(). Figure 10 is the example codes of a base_interrupt_handler class. Details of each receiving process will be covered in the subsequent sections.

```
// Base interrupt handler transactor
class base_interrupt_handler extends vmm_xactor;
    // Message Logger
    vmm_log                         log;
    // Service triggering mode
    int      service_trigger_mode;    // 0: monitor signal/event  1: poll  2: both

    extern function new(string instance);   // constructor
    // Interrupt monitoring method
    extern virtual task wait_for_interrupt_t();

    // Interrupt servicing step 1: Read from status register(s)
    extern virtual task read_status_t(string reg_name, ref base_status rcvd_status[$]);

    // Interrupt servicing step2 :  Compare against expected status
    extern virtual task check_status(base_status checked);

    // Place holders for other related processes
    extern virtual task clear_status_t(string reg_name);  // clear DUT status register
    extern virtual stat_id_e get_status_id (string status_name);  // query for status_id

    // Complete interrupt service which contains all steps
    extern virtual task process_interrupt_t();
```

**Figure 10 Base Interrupt Handler**

### *5.2 Interrupt Monitoring Process*

There are two ways to determine when to initiate an interrupt service. One is by monitoring an interrupt signal or event, the other one is by polling at a regular time interval. The first option requires DUT to provide an interrupt indication. The second option can be used whether or not DUT provides an interrupt indication. Base interrupt handler can support both options. The ***service_trigger_mode*** field of the base_interrupt_handler class is for the purpose of choosing the option(s).

In the case when DUT provides interrupt indicating signal or event, such indication usually can be disabled by software through interrupt masking register. In order to verify that DUT responds with proper interrupt indication, verification environment needs to enable interrupt initially (for example, in the VMM configure DUT phase). Since the interrupt signal or event is DUT specific, the wait_for_interrupt_t() has to be implemented in the derived interrupt handler class where DUT's interface is properly defined. Figure 11 shows an example implementation of wait_for_interrupt_t() and how it takes into account the different service_mode.

```
// Exampe DUT interrupt interface
interface interrupt_if (input clk);
   wire   interrupt;
   clocking cb @(posedge clk);
    input interrupt;
   endclocking: cb
   modport DUT (output interrupt);
   modport TB (clocking cb);
endinterface

// DUT specific interrupt handler derived class
class my_interrupt_handler extends base_interrupt_handler;
   // Virtual interface
   virtual interrupt_if.TB   interrupt_port;

   // Constructor
   function new ( …, my_if.TB  in_port);
       super.new(…);
       interrupt_port = in_port;
   endfunction

   //  Interrupt monitoring process
   virtual task wait_for_interrupt_t();
       case (service_mode)
            0:  begin    // Sampling signal/event
                while (interrupt_port.interrupt!==1) repeat(10) @interrupt_port.cb;
            end
            1:  begin   // Polling
                repeat (cfg.polling_interval) @interrupt_port.cb;
            end
            2:  begin   // Sampling AND polling
                fork
                  begin while (interrupt_port.interrupt!==1) repeat(10) @interrupt_port.cb; end
                   begin repeat (cfg.polling_interval) @interrupt_port.cb;
                join_any
             end
          endcase
   endtask
```

**Figure 11  wait_for_interrupt_t() implementation example**

### *5.3  Interrupt Status Accessing Process*

When an interrupt servicing criterion is satisfied, whether it is triggered by a signal or an event, or a specific time period has elapsed, interrupt handler will start register accessing process. The accesses include reading from the interrupt status register and then clearing the interrupt status register.  If interrupt status register is read-to-clear type, then single read to the register will accomplish both tasks. If interrupt status register is write-one-to-clear type, then a write access needs to be issued after the read access to clear the register.

In order to make ***read_status_t()*** and ***clear_status_t()*** as generic as possible, no implementation specific argument is passed to these two tasks. When we create a derived interrupt handler in a specific verification environment, the derived interrupt handler also needs to add processes to setup the connection between itself, the register accessing transactor, as well as the registers it needs to access. The ***read_status_t()*** and ***clear_status_t()*** in the derived interrupt handler will then be able to physically access DUT's registers.

Figure 12 presents an example ***read_status_t( )*** of a derived interrupt handler. The example uses an example register model which consists of a register field class, register class and a register factory class. These example register modeling classes are also shown in Figure 12. With the example register modeling structure, the derived interrupt handler only needs to pass in the handle of register factory through its constructor and be able to setup the connection between itself and the registers.

```
class my_interrupt_handler extends base_interrupt_handler;
    reg_factory  reg_factory_inst;

    function new(.. reg_factory in_reg_factory,…);    // Constructor
      super.new(..);
      if (in_reg_factory!=null) reg_factory_inst = in_reg_factory;
   endfunction

   virtual task read_status_t(string reg_name, ref base_status rcvd_status[$]);
      bit[63:0] rdata;
      base_register  v_reg;
      base_status     v_status;
      v_reg = reg_factory_inst.get_reg_by_name(reg_name);
      reg_factory_inst.read_reg_by_name_t(reg_name,rdata);
      foreach(v_reg.fields[i]) begin
         if (v_reg.fields[i]>0) begin
            v_status = new;
            v_status.status_name = v_reg.fields[i].fname;
            v_status.status_value = v_reg.fields[i].value;
            v_status.status_id       = get_status_id(reg_name,v_reg.fields[i].fname);
            rcvd_status.push_back(v_status);
         end  // if
      end // foreach
   endtask
endclass

// ---------      Example Register Model  -----------------
class base_register;                                  class base_reg_field;
  string name;                                           string      fname;
  base_reg_field  fields[$];                             bit[63:0]  value;
   …                                                      …
endclass                                              endclass

class reg_factory;
   base_register    registers[$];

   // Return handle of a named register, if not exist, return NULL
   virtual function base_register get_reg_by_name(string name);
      get_reg_by_name = null;
      foreach(registers[i])
         if (registers[i].name == name) get_reg_by_name = registers[i];
   endfunction

   // Register read Access
   virtual task read_reg_by_name_t(string name, output bit[63:0] rdata);
      base_register  v_reg = get_reg_by_name(name);
      if (v_reg != null)  read(v_reg,rdata);  // Read register
   endtask
```

**Figure 12 Example read_status_t() and Register Model**

In the derived ***read_status_t()*** task, it calls base_interrupt_handler's built-in function ***get_status_id()***. This function maps the register and field name strings to the enumerator value of stat_id_e If ***status_id*** field of base_status class does not use enumerator type, this mapping function needs to be modified to return integer value of ***status_id***.

So far, we have been discussing about reading from one status register. For some designs, there are several status registers. In this case, during ***read_status_t()***, each status register needs to be accessed, and each read return status will be put in the rcvd_status queue. All the received status will later on be checked against their corresponding expected status. There are also designs which have registers associated with interrupt status, such as, interrupt count register or interrupt address register. Accessing of these additional registers will also be conducted in the ***read_status_t()*** process. Figure 13 shows the pseudo codes for dealing with additional registers.

```
// Accessing Multiple Status Registers
task read_status_t (string reg_name, ref base_status rcvd_status[$]);
    base_status   rcvd_main_status, rcvd_sub1_status, rcvd_sub2_status;

    // reg_name is the Main register's name
    rcvd_main_status = read_from_register_and_build_status(reg_name);
    rcvd_status.push_back(rcvd_main_status);
    rcvd_sub1_status = read_from_register_and_build_status("sub1_status_register");
    rcvd_status.push_back(rcvd_sub1_status);
    rcvd_sub2_status = read_from_register_and_build_status("sub2_status_register");
    rcvd_status.push_back(rcvd_sub2_status);
 endtask

 // Accessing Status Associated Registers
task read_status_t (string reg_name, ref base_status rcvd_status[$]);
    base_status   loc_rcvd_status;
    string         address_reg_name;
    bit[63:0]      address;

    loc_rcvd_status = read_from_register_and_build_status(reg_name);
    address = read_from_interrupt_address_register();
    loc_rcvd_status.address["address_reg_name"] = address;
endtask
```

**Figure 13 Pseudo codes of accessing multiple status registers or status associative registers**

### *5.4 Check interrupt status*

A self-checking interrupt testing needs to check whether DUT has stored proper information of the interrupt information. This is a generic process which applies to every interrupt handling process and hence it is necessary for the base interrupt handler to have a built-in task to compare the actual read return status against the expected status, i.e., ***check_status().***

In section 4.2, we have discussed about scoreboard expected status at the time when interrupt conditions are generated. At the receive side, in order to access expected interrupt status, the interrupt handler needs to pass in the handle of scoreboard. The scoreboard handle can be passed through interrupt handler's constructor so that the ***check_status()*** task can access it. The ***check_status()*** task has DUT's read back status value as its input argument. It compares the input status against the expected status entry on the scoreboard. Figure 14 shows an example ***check_status()*** task of a derived interrupt handler class.

```
class my_interrupt_handler extends base_interrupt_handler;
   base_scoreboard    sb;
   function new(.., base_scoreboard  in_sb, …);
         if (in_sb != null) sb = in_sb;
         …
   endfunction

   virtual task check_status(base_status checked);
      // Built-in task of base_scoreboard: check (entry, statusQ_id)
      sb.check(checked,checked.status_id);
   endtask
endclass
```

**Figure 14 Example check_status() code**

The above example assumes that the base_scoreboard class has a built-in *check()* task. The task takes the input status (i.e.,checked) and compares it against the first item of the scoreboarded status queue whose queue id equals "checked.status_id".

Each time an interrupt is triggered, the check_status() routine examines each set status bit and performs the following procedures.

- If there are no status bits set, an error is asserted
- If there are only status bits associated with disabled or masked status bits then an error is asserted
- If the scoreboard is empty for the associated interrupt set status bit, an error is asserted
- If the scoreboard is NOT empty for the associated interrupt set status bit, then a scoreboard item is removed. Additionally, a functional coverage point for the status bit is updated.

### *5.5 Complete Interrupt Servicing Process*

A complete interrupt service process consists of one or more iteration of the processes described in previous sub-sections. A built-in virtual method ***process_interrupt_t()*** is created in the base interrupt handler class to allow user to extend and put in the actual implementation.
Having this task, we will be able to create a generic ***main()*** process in the base interrupt handler class. Figure 15 presents the generic ***main()*** routine of the base_interrupt_handler.

```
task base_interrupt_handler::main();
  fork
    super.main();
  join_none

  while (1) begin
    wait_for_interrupt_t();
    process_interrupt_t();
  end
endtask
```

**Figure 15 Generic main() process of base interrupt handler**

Figure 16 shows an example process_interrupt_t() task of a derived interrupt handler.

```
class my_interrupt_handler extends base_interrupt_handler;
   …
   virtual task process_interrupt_t();
     base_status   v_status[$];
     // read from INT_STAT register, return status objects
     read_status_t("INT_STAT",v_status);
     foreach(v_status[i])
       check_status(v_status[i]);
     clear_status_t("INT_STAT");
  endtask
endclass
```

**Figure 16 Example process_interrupt_t()**

### *5.6 End-of-test interrupt status checking*

Most of the time, interrupt handler is free running and interrupt process is triggered automatically by the DUT interrupt signal or programmable time interval. But sometimes, especially at the end of simulation, we may want to manually initiate reading and checking on interrupt status registers to see if there is any unexpected interrupt happened. In VMM based verification environment, this check can be conducted in the cleanup phase.

The end-of-test interrupt checking procedure performs the following:

- Wait for a drain delay to ensure that no interrupt routine tasks are still executing.
- Check that at least one interrupt was processed for each of the testbench generated interrupts for all enabled interrupts.
- Check that all statuses are cleared for all enabled interrupts.
- Check that the interrupt status bits are set or cleared based on whether or not the testbench generated for associated interrupts on disabled (or masked) interrupts. Note that some of these bits may have been cleared due to the interrupt polling or other interrupts trigging the interrupt service routine.

- Check that the interrupt service routine cleared statues for interrupts.

Since the free running interrupt handler has already been stopped when simulation gets into cleanup phase, so we will need to call ***process_interrupt_t()*** explicitly to perform the check as shown in Figure 17.

```
class my_env extends vmm_env;
   my_interrupt_handler   int_hdlr;

   task cleanup();
      int_hdlr.process_interrupt_t();
      …
   endtask
endclass
```

**Figure 17   Check interrupt status in cleanup phase**

## 6.0 Fast Posting and Slow Checking

In most cases, interrupt is triggered much faster than it is serviced, especially when the servicing is done through front-door register accesses. This is because the time it takes to detect an interrupt and then access registers to get the status information usually is much longer than the time it takes for an interrupt to be triggered.

As we have discussed in section 4, expected interrupt status are posted on scoreboard whenever an interrupt condition is generated. Therefore it is very often that when an interrupt servicing routine is initiated, there have already been several expected status entries posted on the scoreboard. Since the interrupt source is not taken away during the interrupt servicing process, there could have more interrupt stimulus being generated from the transmit side and hence more expected status entries could be put on the scoreboard as the service is ongoing. So generally speaking, interrupt testing is a fast posting and slow checking process.

In this section, we will briefly discuss about some interrupt triggering situations and the schemes for handling extra scoreboarded status entries using our proposed interrupt handling mechanism.

### *6.1 Interrupt from multiple sources*

This is the case when multiple interrupt sources are triggered from the transmit side. The first happened interrupt will set the interrupt signal and initiate the service routine. When the service routine reads the status register, the status of that particular interrupt source should have been set. Meanwhile, the status of other sources may also be set if by the time the status register is read those sources have already propagated through the interrupt logic. As we have shown inFigure 16, the ***process_interrupt_t()*** will go through all status fields which are detected set and compare against the scoreboarded entries. Thus multiple interrupt sources can be detected even though checking frequency is much slower than the posting frequency.

### *6.2 Multiple interrupts from same source*

Since interrupt sources are not removed right after an interrupt is triggered, there could have many of the same status entries being posted on scoreboard by the time an interrupt service routine is in process. If we don't need to verify the interrupt counts, then whenever the interrupt status register is read and a particular interrupt status is detected, one scoreboard entry in that interrupt status queue will be removed from the scoreboard. At the end of simulation, during the cleanup phase, there may have entries left in that particular status queue. The *object_id* of the first entry in that queue indicates how many such kind of interrupt has been detected. As long as the *object_id* is greater than 1, we can consider it PASS. If DUT keeps track of interrupt count for that particular interrupt status, then during cleanup, the last scoreboard entry left in that status queue should have *object_id* equal to or greater than the DUT indicated interrupt count.

### *6.3 Single interrupt status from multiple sources*

There may be cases when multiple interrupt sources trigger the same interrupt status. Since the scoreboard queue is indexed using interrupt status information, the expected status objects from these multiple interrupt sources will be put in the same scoreboard queue. Therefore, checking for this situation will be the same as what we described in section 6.2.
If we really want to check that each individual interrupt source does trigger the status, we need to create a testcase which enables only one interrupt source at a time.

## 7.0 Other considerations for interrupt verification

### *7.1 At least two rounds of interrupt service*

In order to completely verify interrupt logic, we need to make sure that interrupt service routine is run through for at least two times. A single round of interrupt service process verifies whether interrupt signal or event has been properly triggered by a single or multiple interrupt sources; whether interrupt status are properly registered. The second round of the interrupt service will verify that whether status register is properly cleared; whether the interrupt signal/event is properly reset and can then be triggered again.

Functional coverage metrics updated by the interrupt monitor contain appropriate metrics to ensure that multiple interrupts occur. The functional coverage "at least" attribute controls the number of times the coverage event needs to occur before the coverage point is hit. By setting the "at least" attribute to something greater than the default value of one ensures that, a particular status bit has been processed multiple times.

### *7.2 Cooperate with other register accessing processes in the verification environment*

Since interrupt servicing involves register accesses, it may co-exist with other processes which also request register accesses. For example, we might have a free running register interface transactor to generate random background register accesses. If there is an interrupt happened,

the interrupt handler will start to issue register accesses. To handle such kind of situation, we can make each register access to request a semaphore before it can be issued to the DUT. Semaphores can be created in one of these two places: register access driver which physically issues read/write requests to DUT; interface between register classes and the register access driver, such as a register factory class.

If semaphore is built into the driver class, accesses issued from interrupt handler as well as the other register accessing transactor will be automatically managed by the semaphore.

If a verification environment has a register factory and the register factory has tasks to pass register object to register access driver, the semaphore can be created in the register factory. As long as all register accesses are done through register factory (i.e., through calling *write_register()* and *read_register()*), nothing needs to be done by the interrupt handler and the other transactor. Figure 18 shows code example for this case.

```
Class dummy_register_factory;
   register_class   reg_Foo;
   register_class   reg_Bar;
   register_driver  driver;
   semaphore        reg_semaphore;
   ….
   task write_register(register_class ireg, int data);
         reg_semaphore.get(1);
         driver.write(ireg.address, data);
         reg_semaphore.put(1);
   endtask

   task read_register(register_class ireg, ref int data);
         reg_semaphore.get(1);
         driver.read(ireg.address, data);
          reg_semaphore.put(1);
    endtask
   ….
endclass
```

**Figure 18  Manager register access order through Semaphore in register factory class**

## 8.0 Conclusion

In this paper, we presented an interrupt verification mechanism that can be implemented early in the verification environment development process. Both transmit side and receive side of interrupt verification process are discussed. VMM based interrupt status base class and VMM based interrupt_handler transactor class are employed in the mechanism. Example codes are provided to demonstrate how to apply this mechanism in a verification environment. In addition, some of the complex interrupt issues, such as verifying fast or slow interrupt handling and masked/unmasked testing, are also addressed in the paper.

This mechanism allows a verification team to conduct interrupt verification with common look and feel. Without such kind of mechanism, doing interrupt verification in some kind of “ad hoc” manner will likely slow down the verification effort and/or inhibit the testbench’s self checking capabilities.

## 9.0 References

Verification Methodology Manual for SystemVerilog, *Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale* Springer 2005