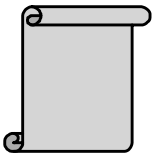
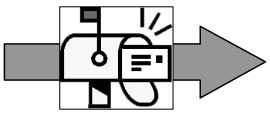
2 VMM TRANSACTIONS AND CHANNELS





This chapter addresses the definition of transactions and channels. Since we use a FIFO as the DUT model, a description of the FIFO controller model and interface is first.

2.1 THE DUT

The design under test (DUT) used throughout the book is a synchronous first-in first-out (FIFO) model as shown in Figure 2.1-1. The model consists of two blocks: the FIFO block and the configuration block. The **FIFO block** is a simple synchronous FIFO controller with PUSH, POP, and RESET commands. Upon a PUSH, the *data_in* is stored into the FIFO internal memory. Upon a POP, *data_out* provides in the same cycle the data off the stack. Five flags are provided: *full, empty, almost_full* and *almost_empty* flags to identify the status of the FIFO, and an *error* flag to identify an erroneous PUSH on full or POP on empty. The active low *reset_n* resets the FIFO to the empty state. The **configuration block** can be used by an external controller to configure the levels of the *almost_full* and *almost_empty* flags. However, this design can work without such external controls because a hard reset configures those levels to a default mid-level value.

In the design of the DUT we have a choice in port style for the definition of the interconnections: SystemVerilog interfaces or ports-only a la Verilog'95. This decision is irrelevant to the testbench design because the top-level connections of a SystemVerilog interface instance to the DUT instance can accommodate either style. Thus, the ports of a DUT can be specified as either ports-only a la Verilog style, or with interfaces a la SystemVerilog style as shown in Figure 2.1-2. Some designers use the SystemVerilog *interface* definition in the RTL design. Others restrict the design to the Verilog style with individual port signals, instead of grouping the signals with SystemVerilog interfaces**.** If SystemVerilog interfaces are not defined, it is necessary for the verification engineer to define such interfaces at the top-level block. This is because VMM requires that SystemVerilog interfaces representing the DUT port interconnections be made available to the testbench. This facilitates the connections to the verification environment defined in classes through the use of virtual interfaces. We selected for our DUT FIFO the use of SystemVerilog interfaces because the SystemVerilog *interface* abstracts the communication across several modules, but a port-only style would have been acceptable since the port style has no significant impact on the testbench – the changes are only at the top level interconnection of the DUT instance to the SystemVerilog interface instance.

The SystemVerilog LRM states "*The interface construct in SystemVerilog was specifically created to encapsulate the communication between blocks. By encapsulating the communication between blocks, the interface construct also facilitates design reuse.* Note that other items can be specified within interfaces include *clocking* blocks, *modports*, assertions, and *covergroups*. An *interface* may also have tasks and assertions associated with the operation of the signals of the interface. VMM (rule 4-9) recommends that the definition of tasks associated with the *interface* be located in *classes* and *subclasses* (typically in command transactors), not defined in the interface. This is because interfaces are not object-oriented, and functions and tasks cannot be defined as virtual. Thus, they cannot be redefined to behave differently, such as inject errors or adapt to a new algorithm. Examples of tasks include a *push_task*, a *pop_task*. Interface assertions relate to the properties or timing relationships of those signals. As a methodology, an interface *should* capture assertions related to its signals. [1]

Figure 2.1a demonstrates that the FIFO model includes the two separate blocks, each with its own SystemVerilog interface: the **FIFO block** with the the *fifo_if* interface for the normal operation of the FIFO device, and the **configuration block** with the *fifo_csr_if* interface for the configuration

[1] From VMM Chapter 3 on assertions "Applying assertions to external interfaces treats the DUT as a black box. It is concerned with the correct function of the design, regardless of its implementation. "

of FIFO levels. The configuration block includes configuration registers to be setup by the environment during DUT initialization.

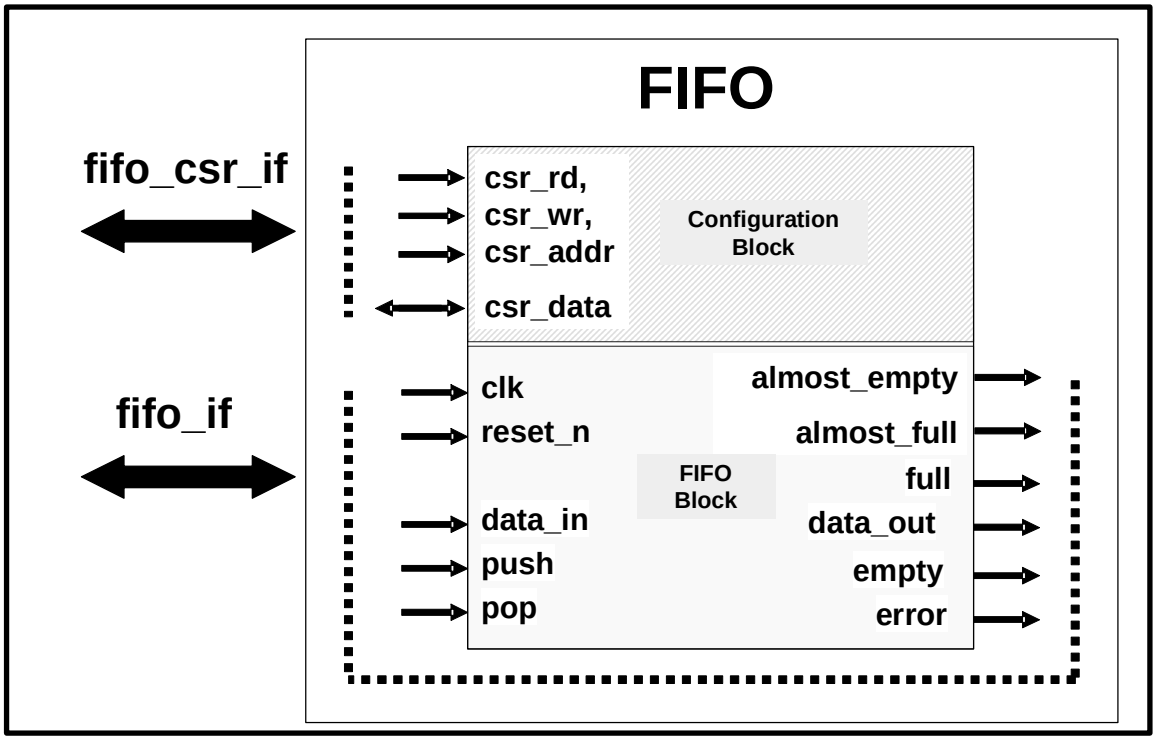


Figure 2.1-1 FIFO Interfaces (*fifo_if* and *fifo_csr_if*)

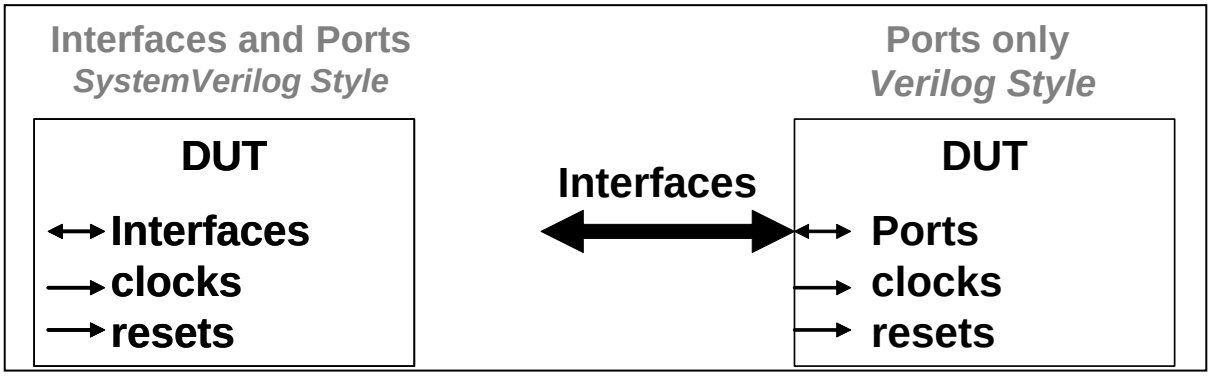


Figure 2.1-2 Port Styles of DUT Ports

Various type definitions are used throughout the design and the testbench, and are specified in the *fifo_pkg.sv* package as shown in Figure 2.1-3.

```
package fifo_pkg;
  timeunit 1ns;
  timeprecision 100ps;
  `define TOP fifo_tb
  typedef enum {PUSH, POP, PUSH_POP, IDLE, RESET} fifo_scen_e;
  typedef enum {PUSH_MODE, POP_MODE} mode_e;
  typedef enum {PASSED, FAILED} fifo_status_e;
  typedef enum {DONE_GEN, DONE_BFM} notification_e;
  parameter BIT_DEPTH = 4;
  parameter FULL = 16;
  parameter WIDTH = 32;
  typedef   logic [WIDTH-1 : 0] word_t;
  typedef   logic[31:0] wword_t;
  typedef   logic [WIDTH-1 : 0] wire_word_t;
  typedef   word_t [0 : (2**BIT_DEPTH-1)] buffer_t;
  parameter            ALM_EMPTY_REG = 2'b00;
  parameter            ALM_FULL_REG = 2'b01;
endpackage : fifo_pkg
```

Figure 2.1-3 Common Type and Parameter Definitions (*ch4_fifo/fifp_pkg.sv*)

Figure 2.1-4 demonstrates the FIFO interface with the *modports* and *clocking blocks* used throughout the design and the testbench.[2]

```
interface fifo_if(input wire clk,
                  input wire reset_n);
   timeunit 1ns;
   timeprecision 100ps;
  import fifo_pkg::*;
  logic  push; // push data into the fifo
  logic  pop;  // pop data from the fifo
  wire  full;  // fifo is at maximum level
  wire  empty; // fifo is at the zero level (no data)
  logic almost_empty, almost_full;

  logic  error; // fifo push or pop error
  word_t data_in;
  wire_word_t data_out;
  parameter HOLD_TIME=3;
  parameter SETUP_TIME = 5;

  clocking slave_cb @ (posedge clk);
   default input #5ns output #HOLD_TIME;
     output empty, full, data_out, error;
     input  data_in, push, pop;
  endclocking : slave_cb
```

[2] VMM recommendations in the use *modport* and *clocking block* are defined in VMM Rule 4-8, Rule 4-9, Rule 4-11, Rule 4-12

```
clocking driver_cb @ (posedge clk);
   default input #SETUP_TIME output #HOLD_TIME;
     input  empty, full, data_out, error;
     output data_in, push, pop;
  endclocking : driver_cb

  // FIFO DUT view of the interface
  modport fslave_if_mp (clocking slave_cb,
                        output empty,
                        output full,
                        output data_out,
                        output error,
                 output almost_empty,
                 output almost_full,
                        input  data_in,
                        input  push,
                        input  pop);

  // FIFO driver view of the interface
   modport fdrvr_if_mp (clocking driver_cb);

// FIFO Monitor view of the interface.
 clocking mon_cb @ (posedge clk);
   default input #SETUP_TIME output #HOLD_TIME;
     input  empty, full, data_out, error;
     input data_in, push, pop;
  endclocking : mon_cb

  modport fifo_mon_if_mp (clocking mon_cb);

endinterface : fifo_if
```

Figure 2.1-4 FIFO Interface (file *ch4_fifo/fifo_if.sv*)

Figure 2.1-5 represents the FIFO configuration register interface.

```
interface fifo_csr_if (input clk, reset_n);
  timeunit 1ns;
  timeprecision 100ps;
  logic csr_rd, csr_wr;  // configuration read, write
  logic [1:0] csr_addr;  // configuration reg address
  logic [4:0] csr_data;  // configuration reg data

modport fifo_ctl_mp (input csr_rd, csr_wr, csr_addr,
                     inout csr_data);

endinterface : fifo_csr_if
```

Figure 2.1-5 FIFO Configuration Register Interface (file *ch4_fifo/fifo_csr_if.sv*)

Partial code for the FIFO DUT is shown in Figure 2.1-6, with the full code available in the download. The FIFO port connections include the clock, the reset, and the FIFO interface with the slave *modport fifo_if.fslave*, and the configuration interface with the control *modport. fifo_csr_if.fifo_ctl_mp*

```
module fifo (input clk,
             input reset_n,
             fifo_if.fslave_if_mp f_if,
            fifo_csr_if.fifo_ctl_mp f_csr_if
           );
// Note: We're using a modport in the DUT instead of
// an interface to avoid any accidental RTL writes to an input
// This is a good coding guideline.
  timeunit 1ns;
  timeprecision 100ps;
  import fifo_pkg::*;
  logic [4 : 0] wr_addr;
  logic [4 : 0] rd_addr; // read fifo address
  typedef enum {NONE, PUSH, POP, PSPP} push_pop_e;
  buffer_t buffer; // fifo storage
  parameter shiftVal = int'(2**BIT_DEPTH);

  logic [4:0] fifo_level, csr_almost_empty, csr_almost_full;
  // Push on full with no pop error
  property p_push_error;
    @ (posedge clk)
       not (f_if.push && f_if.full && !f_if.pop);
  endproperty : p_push_error
  ap_push_error_1 : assert property (p_push_error);

 // *** SEE DOWNLOAD FILE FOR FULL CONTENT ***
  …
endmodule : fifo
```

Figure 2.2b FIFO at RTL Level (*ch4_fifo/fifo_rtl.sv*)

2.2 TRANSACTION

VMM defines a transaction as *an operation on an interface. A transaction can be abstract and high-level, such as the reliable transmission of a TCP packet, or physical, such as a write cycle on a APB. Interconnect.* Thus, a transaction represents the job to be performed, such as Read / Write / Idle. Figure 2.2-1 is a review of the testbench architecture to demonstrate the emphasis of where in the environment the transactions and the channel reside. Transactions are typically randomized and *put* into a channel by the generator. The transactions are then extracted (via the *get* method) by the command transactor for processing (more on this in Chapter 3).

Transactions are implemented with a class extended from *vmm_data* base class.

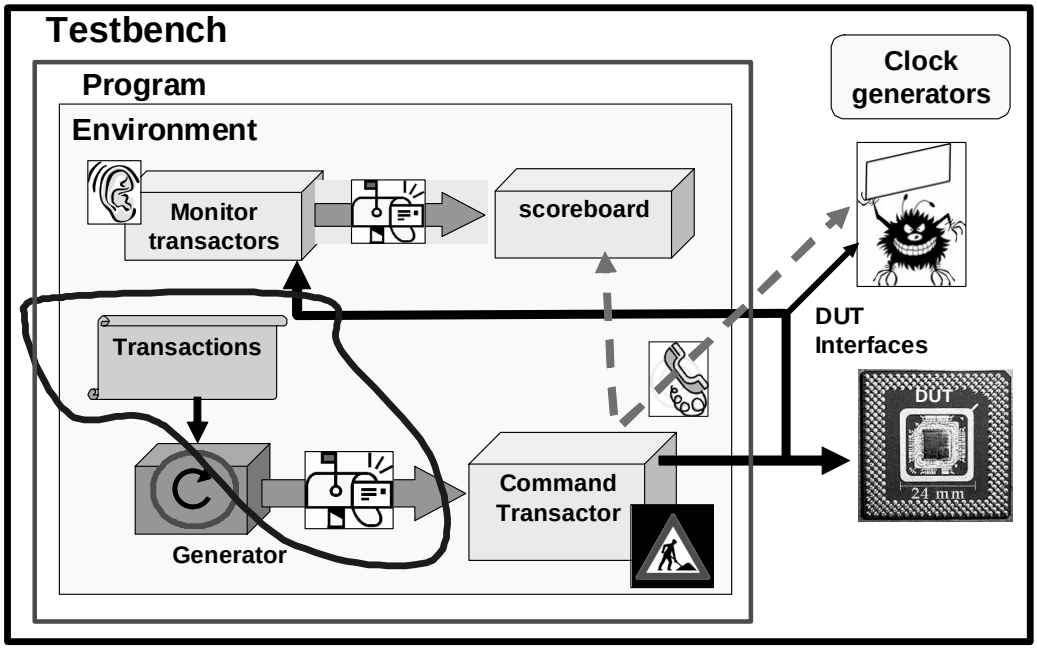


Figure 2.2-1 Testbench Architecture with Emphasis on Transactions and Channels

What's in a *transaction* class? The UML diagram, shown in Figure 2.2-2, summarizes the important (but not all) elements of the class. Basically, a transaction class is an extension to base class *vmm_data*, and includes:

1. A set of properties, also known as class variables. The properties characterized with the **rand** qualifier can be randomized with the **randomize** function of the class instance.
2. A set of methods to support the class. The *copy* function is very important, and is discussed further down.

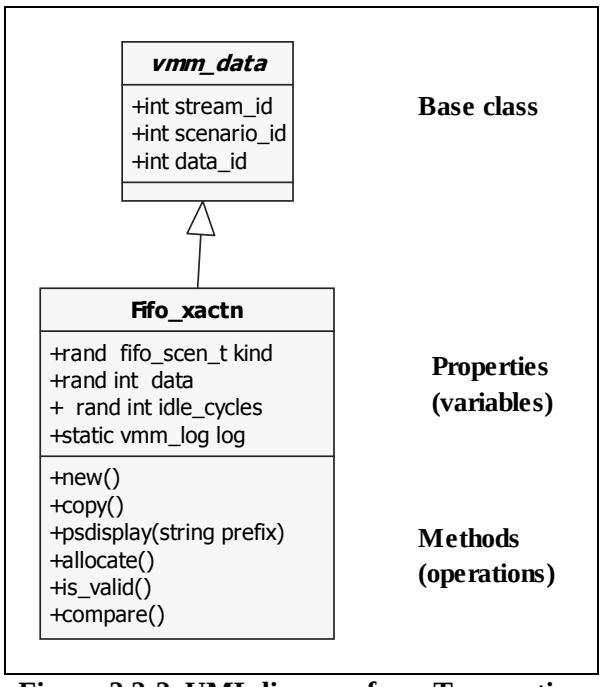


Figure 2.2-2 UMLdiagram for a Transaction

2.2.1 Extension to *vmm_data*

Using the FIFO example, the first line of the transaction class definition is:

```
class Fifo_xactn extends vmm_data;
```

vmm_data is the base class type used to build data models that flow from transactors to transactors over time, such as a transaction model that flows from a generator to a transactor through a channel.[3] In contrast, configuration descriptors are example of classes that should not be derived from *vmm_data*. A multi-dimensional look-up table in a scoreboard is another example. Why is transaction class extended from *vmm_data*? Because *vmm_data* is part of the VMM framework and it provides several support methods that operate on objects of that class. In addition, transactions are usually passed across the environment through VMM Channels, and a vmm_channel is a strongly typed SystemVerilog queue that can only carry *vmm_data* and its derivatives.

2.2.2 Properties and Constraints

Following the class declaration line you define the properties or variables that describe the transactions. Variables that need to be class randomized (e.g., *class_instance.randomize())* must be qualified with the *rand* attribute.[4] Properties qualified with the *rand* will be randomized with the automatically generated *atomic* or *scenario* generators via macros, which are discussed in subsequent chapters. Types of objects to be randomized include address, data, modes, etc.

2.2.2.1 Properties

Figure 2.2.2.1-1 represents the properties and constraints for the FIFO model. The variable *kind* identifies the possible instructions to be executed by the transactor. For readability the variable *kind* should be of an enumerated data type. In our design, that type is

```
typedef enum {PUSH, POP, PUSH_POP, IDLE, RESET} fifo_scen_e;
```

Ideally, you have to identify the variable *kind* to represent all possible independent tasks that the transactor will execute. Because design requirements change, it is often difficult to predict all the needed enumerations. In the case of the FIFO design, the new requirement is the need to create a transaction that specifies a PUSH with a data error. As an afterthought, we considered adding to the original type definition *fifo_scen_e* the enumeration PUSH_DATA_ERR to handle that case. However, we wanted to demonstrate real life change in requirements and the capability of adding changes to the execution of transactions through the use of factories and callbacks. These are explained in subsequent chapters, but essentially, the approach we took is to substitute an IDLE instruction with a PUSH instruction (as defined by the transaction) with bit errors. Not all IDLE instructions will have this substitution, but rather, this is determined on the probability of randomized variable to inject or not inject this error when the transaction calls for an IDLE instruction.

In this model, the *data* represents the data to be stored into the FIFO via the *data_in* port; the *idle_cycles* represents the number of idle cycles to be executed during the IDLE transaction; and *xactn_time* is a variable that is intended to be used as a time stamp for debug purposes. An alternative to the *xactn_time* is the use the timestamp built-in into the *vmm_notify* notifications in *vmm-data* STARTED and ENDED. Those concepts are addressed in Chapter 7.

---

[3] VMM Rule 4-55: Data and transaction model classes shall be derived from the *vmm_data* class.

[4] VMM Rule 4-59: All class properties corresponding to a protocol property or field shall have the *rand* attribute.

The *log* variable is of *vmm_log* class that implements an interface to the message service. VMM requires that the *log* property of a transaction be declared *static*.[5] By default all class properties are automatic and are created and destroyed on the fly. However, a messaging service for transactions should be unique to provide consistent messaging and to avoid a potential simulation performance impact. There will be thousands of object instances created and destroyed throughout a simulation, and it is needless to create so many of messaging service agents.

```
class Fifo_xactn extends vmm_data;
  import fifo_pkg::*;
  rand fifo_scen_e kind;
  rand word_t  data;  // in : data to push
  rand int idle_cycles;
  time xactn_time;
  static vmm_log log = new("Fifo_xactn", //name
                           "class");[6]    // instance
 constraint cst_idle {
   idle_cycles inside {[1:3]};
  }

  constraint cst_xact_kind {
    kind dist {
      PUSH := 400,
      POP := 300,
      PUSH_POP :=200,
      IDLE := 300,
      RESET := 1
    };
  } // cst_xact_kind
…
```

Figure 2.2.2.1-1 Properties of FIFO Transaction (*ch4_fifo*/fifo_xactn.sv)

From *vmm_log*, the format of the function is:

```
function vmm_log::new(string name,
                      string instance,
                      vmm_log under = null);
```

That creates a new instance of a message service interface, with the specified interface name and instance name. In our example, we used:

```
static vmm_log log = new("Fifo_xactn", "class");
```

The log is displayed with the application of a messaging macro, such as *`vmm_note*.[7] If we modify the *new* function of the *fifo_xactn* class with the addition of a *`vmm_note* as shown in Figure 2.2.2.3, then the note is displayed during the simulation run at every allocation of the transaction object. For example,

```
Normal[NOTE] on Fifo_xactn(class) at              0.00 ns:
    New fifo_xactn
```

[5] VMM Rule 4-58 All data classes shall have a public static class property referring to an instance of the message service interface.

[6] VMM Example 4-33. Declaring and Initializing a Message Service Interface

[7] VMM Appendix A, *vmm_log*, Table A-2

VMM is a flexible framework that adapts to users' requirements. For example, VMM provides the capability to globally set the message formatter to the specified message formatter instance. [8] With our changes to the log format, the result of the above *`vmm_note* would yield the following:

```
0.00 ns Fifo_xactn [Normal:NOTE] | New fifo_xactn[9]
```

2.2.2.2 Constraints

The constraints determine the legal (and reasonable) values that can be assigned to the random variables. In our example we constrained the number of idle cycles and the distributed value of the variable *kind.*

2.2.2.3 Methods

There are three groups of pre-defined *vmm_data* methods that need to be overloaded by the user in the definition of a class derived from *vmm_data.*[10] These include:

1. Basic methods: *new*(), *allocate*(), *copy*(), *compare()*
2. Debug method: *psdisplay*()
3. Physical handling methods: *byte_pack(), byte_unpack(), max_byte_size().* Refer to the VMM book for usage of methods.

Figure 2.2.2.3 demonstrates the methods used for the FIFO transaction class, including the *new*, *copy*, *allocate*, and *psdisplay*. Except for the *new*, all methods in our code have a prototype and **extern** definitions to enhance readability. This use of **extern** is a general software recommendation. Note that if a method has arguments, then the arguments can have default values at the prototype class definition level, but not in the external implementation level. We specified the default values of a method at the prototype level.

```
class Fifo_xactn extends vmm_data;
…
  function new();
    super.new(this.log);
  `vmm_note(this.log, "New fifo_xactn");
  endfunction : new
  extern virtual function string psdisplay(string prefix = "");
  extern virtual function vmm_data copy(vmm_data to=null);
  extern virtual function vmm_data allocate();
  extern virtual function bit compare(vmm_data   to,
                                      output string diff,
                                      input int kind = -1
                                      );
endclass:Fifo_xactn
```

Figure 2.2.2.3 Transaction Methods in the Class (*ch4_fifo/fifo_xactn.sv*)

---

[8] Throughout this book we modified the default message formatter using the function *set_format (vmm_log_format fmt)*, VMM Appendix A, *vmm_log*. Chapter 8 provides an explanation of the changes.

[9] VMM page 372, Table A-2. Message Type and Severity for Shorthand Macros defines other macros include: *vmm_fatal(), vmm_error(), vmm_warning(), vmm_note(), vmm_trace(), vmm_debug(), vmm_verbose(), vmm_report(), vmm_command(), vmm_transaction(), vmm_protocol()* and *vmm_cycle()*

[10] VMM Rule 4-76. All classes derived from the *vmm_data* class shall provide implementations for the *psdisplay(), is_valid(), allocate(), copy()* and *compare()* virtual methods. VMM Appendix A, *vmm_data* specifies the base methods.

2.2.2.3.1 new()

In the model, the *new* does not have any argument. The *new* function must call *vmm_data::new* and pass the log object that was statically allocated.[11]

```
function new();
    super.new(this.log);
endfunction : new
```

2.2.2.3.2 allocate()

The *allocate* function is used for the factory pattern, and is explained in Chapter 5. The format is as shown in Figure 2.2.2.3.2.

```
function vmm_data Fifo_xactn::allocate();

    Fifo_xactn fifo_xactn_local=new();
    allocate = fifo_xactn_local;
endfunction : allocate
```

Figure 2.2.2.3.2 *allocate* Function

2.2.2.3.3 copy()

The *copy* function is necessary to create a copy of the source transaction descriptor. The *copy* function allows the creation of a duplicate of the transaction to be sent to a channel while maintaining the original version pristine and ready to be modified by the original transactor that modifies the transaction. For example, a transaction generator (atomic, scenario, or user-defined) may need to update a transaction based on its previous value (e.g., increment the address), and then send a copy of that modified transaction to a channel, thus keeping the versions stored in the channel separate and distinct form the version being modified by the transaction generator. The VMM recommends the implementation style shown in Figure 2.3.2.3.3.[12]

```
function  vmm_data Fifo_xactn::copy(vmm_data to);
  Fifo_xactn cpy;

  if (to !=null) begin
    if (!$cast(cpy, to)) begin
      `vmm_fatal(log,
                 "Attempting to copy a non fifo_xactn instance");
       return;
     end
  end else cpy =new;
  super.copy_data(cpy);
  cpy.kind = this.kind;
  cpy.data = this.data;
  cpy.idle_cycles = this.idle_cycles;
  copy = cpy;
endfunction : copy
```

Figure 2.3.2.3.3 Copy Function

---

[11] VMM Appendix A, *vmm_data*, function *new()*.

[12] VMM Appendix A, *vmm_data*, Example A-7. Proper Implementation of the *vmm_data::copy()* Method Also, see page 385 for the model.

2.2.2.3.4 compare()

The compare function is used to dynamically compare the predicted response to the observed response.[13] An example of the *compare* function is shown in Figure 2.2.2.3.4. For our simple FIFO model, we did not make use of this function.

```
function bit Fifo_xactn::compare(vmm_data   to,
                                 output string diff,
                                 input int kind
                                 );
begin
   Fifo_xactn cmp;
   string tmp_str;

   compare = 1; // Assume success by default.
   diff    = "";

   // Cast assign the vmm_data handle to an Fifo_xactn handle
   if (!$cast(cmp, to)) begin
      `vmm_fatal(this.log,
        "Attempting to compare to a non Fifo_xactn instance");
      compare = 0;
      diff = "Cannot compare non Fifo_xactn to Fifo_xactn";
      return;
   end

   // Compare the individual data members, and set compare to 0 and
   // the "diff" text string to "xxx field mismatched" on failure.

   if (this.data !== cmp.data) begin
     compare = 0; diff = {diff, "Data Mismatch "}; end
   if (compare == 0)begin
     diff = {"Fifo_xactn objects are not identical. Mismatched
field(s) is(are):\n", diff};
   end
end
endfunction
```

Figure 2.2.2.3.4 *compare* Function (/ch4_fifo_fifo_xactn.sv)

2.2.2.3.5 psdisplay(string prefix = "")[14], [15]

This is a useful function that returns an image of the current value of the transaction or data in a human-readable string format. We used this function to display a message and the value of the current transaction *kind* variable. The prototype of the function is:

```
virtual function string psdisplay(string prefix = "");
```

---

[13] For application example, see VMM Example 5-49. Checking In-Order Response for Multiple Output Ports

[14] VMM Rule 4-45 . All simulation messages shall be sent through the message service.

[15] VMM Rule 4-76 . All classes derived from the *vmm_data* class shall provide implementations for the *psdisplay(), is_valid(), allocate(), copy()* and *compare()* virtual methods.

. The body of the function with a user defined message for the FIFO transaction is:

```
function string Fifo_xactn::psdisplay(string prefix);
    $sformat(psdisplay,
             "%s Fifo Xaction %s \n",
             prefix, this.kind.name());
endfunction : psdisplay    // ch4_fifo/ fifo_xactn.sv
```

In a transactor, one can apply this function as shown below:

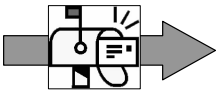
```
`vmm_note(this.log,
  $psprintf("Got a new fifo xaction from in_channel %s ",
            fifo_xactn_0.psdisplay())); // ch4_fifo/ fifo_cmd_xactor.sv
```

`vmm_note is a macro that provides a shorthand notation for issuing single-line note messages. During simulation, the following message gets printed to the log file. With the default display format, the log has the following look in two lines:

```
Normal[NOTE] on Fifo_cmd_xactor(class) at          1950.00 ns:
    Got a new fifo xaction from in_channel  #0.0.0 Fifo Xaction PUSH
```

With our customized display format, we get the following look in one line:

```
1950.00 ns cmd_xactor [Normal:NOTE] | Got a new fifo xaction from
in_channel  #0.0.0 Fifo Xaction PUSH
```



2.3 CHANNEL

Per VMM, the channel object is the primary transaction and data interface mechanism used by transactors. [16] It can be used with any class that is derived from the *vmm_data* class. In our example, as shown in Figure 2.3, a channel is used to connect the transaction generator to the transactor. It can also be used to connect a bus monitor to the scoreboard.

Basically, a channel is a conduit built for a particular type of data object (derived from *vmm_data*). A channel is a class that holds handles to transaction objects. It is implemented with a bounded or unbounded queue of handles to transaction objects. However, channels can be reconfigured for size.

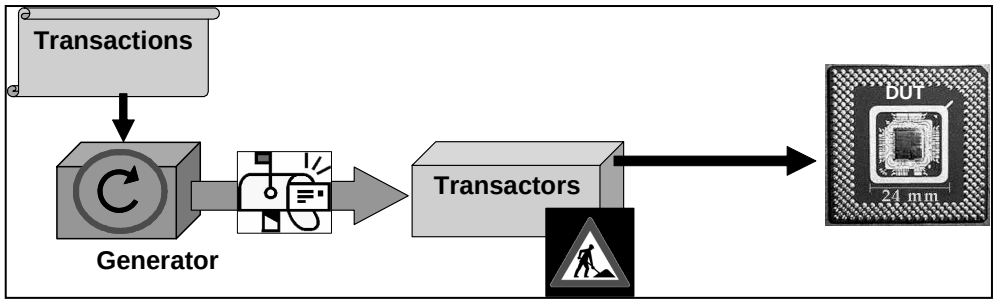


Figure 2.3 Typical Use of Channels

2.3.1 Creation of channels

In the VMM framework, channels are generated with the macro *`vmm_channel (trasaction_class_name).* This macro is typically added in the same file where the specified class is defined and implemented. In our example, this macro is at the end of the file *fifo_xactn.sv*.

[16] VMM Rule 4-111 A channel shall be used to exchange transactions between two transactors.

The macro creates an external class declaration with name *Fifo_xactn_channel*. In our example, we used the following:

```
// This macro declares new derived class named:
// fifo_xactn_channel from vmm_channel
`vmm_channel (Fifo_xactn)
```

2.3.2 Access to Elements in a Channel

VMM framework provides a very rich set of methods that support channels, as shown in UML in Figure 2.3.2. In our FIFO model we made use of the *new, put* and *get* methods. A description of the most commonly used methods is provided below. However, *vmm_channel* supports complex requirements in handling transactions in channels (e.g., out-of-order execution model) with methods that let transactors query the execution progress of a transaction directly from the channel itself. Refer to the VMM book for a description and application of these methods.

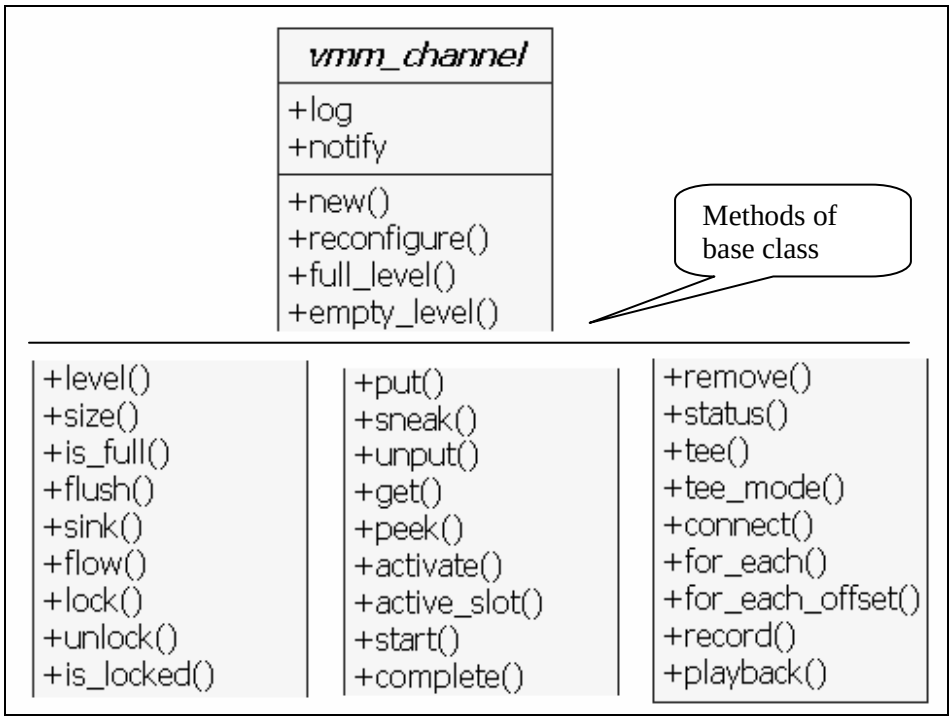


Figure 2.3.2-1 *vmm_channel* Properties and Methods

Transactions are transferred through a channel using the *put* and *get* tasks (provided in the created channel) and the optional use of an offset (without an offset, the default of the task is used). Figure 2.3.2-2 demonstrates the offset relationships of a channel queue. Note that the channel offsets items can be ordered from either end. New transactions are stored by default at the tail of the queue (i.e., -1), and are extracted by default from the head of the queue (i.e., 0). Also note that there are two ways to define the offset:

1. Natural number (e.g., 0, 1, 2, .. n), where "0" is the head of the channel. Thus, an offset of "1" represents the transaction prior to the one at the head of the queue.
2. Negative number (e.g., -1, -2, …-n) where "-1" is the tail of the channel. Thus, an offset of -2 represents the transaction prior (or more recent) to the one at the tail of the channel.

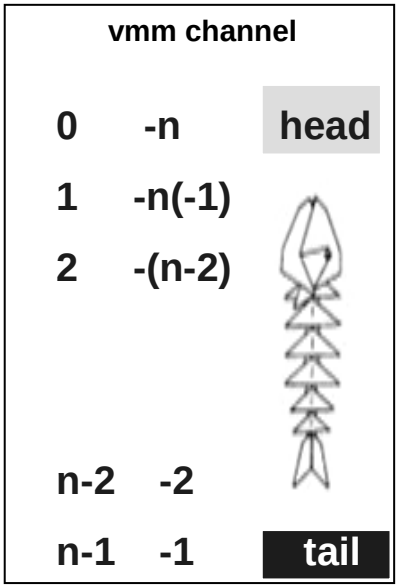


Figure 2.3.2-2 Offset Relationships of a Channel Queue

The created channel class provides several methods to put or get transactions into and out of the channel. These methods include the *put, get, peek,* and *sneak* tasks, as described below. Channels are blocking with the *put* and *get* tasks. If the channel is full, and the consumer is not ready to pull from a full channel, then the producer will be blocked from adding more transactions. The prototype for the *put* method is:

```
task put(class_name obj,
         int   offset = -1); // tail is the default offset
```

The *put* inserts the transaction object at the specified offset. The task blocks if the channel is full.

```
task get(output class_name obj,
         input int offset = 0); // head is the default offset
```

The *get* task retrieves the next transaction at the specified offset. The task blocks if the channel is empty.

Why are the *put* and *get* methods blocking when the maximum level of the channel is reached? Those methods are blocking because this mechanism is a self-regulating control flow mechanism where no transactor goes faster than the slowest of all transactors in a pipeline. The producer of transactions may be capable of producing a large number of transactions at a rate much faster than what can be consumed. However, because the flow control is throttled by the channel, replacing a transactor with a different generation/consumption speed has no effect in modifying the channel code. For example, if the default channel level of one is used, consider the following situations:

1. Channel is empty: the consumer is blocked if it attempts to *get* the transaction. There is nothing to get, thus the blocking is obviously necessary.
2. Channel is full: The producer is blocked if it attempts to *put* a transaction. This mechanism also avoids an unnecessary overload on the simulator resources in filling up the channel with transactions that are not yet needed by the consumer. For example, if a transaction occupies 1000 bytes of resource, and the generator is capable of producing one million transactions, then a nonblocking *put* would put a load of 1GB just for the channel. Instead, a blocking *put* (as implemented in VMM) throttles the production of channels in synchronism with the consumption of those channels within the set bounds of the channel size.

A snippet of code that uses the *get* is demonstrated in Figure 2.3.2-3.

```
class Fifo_cmd_xactor extends vmm_xactor;
  …
  Fifo_xactn  fifo_xactn_0;
  Fifo_xactn_channel in_chan;
  begin : main_loop
    this.in_chan.get(fifo_xactn_0);
     case (fifo_xactn_0.kind)
      PUSH :
        begin
          this.push_task(fifo_xactn_0.data);
        end
  ..
```

Figure 2.3.2-3 Snippet of Code Using *get (ch4_fifo_fifo_cmd_xactor.sv)*

The prototype for the *sneak* method is:

```
task sneak(class_name obj);
```

vmm_channel::sneak() is a non-blocking method to add items to the tail of a channel, as compared to a *put*(), which is blocking. The *sneak*() method simply ignores any pre-configured full level. This can be performed because the channel queue buffer is infinite and not statically allocated.

A question that often comes up is "why do we need the *sneak* method, and where should we use that method"? To answer that question you need to understand how channels handle the storing of transactions. Channels must be sunk; meaning that some other component in the environment must perform a *get* of the transaction to unblock the channel that reached its maximum level (and to allow a *put* of another transaction). Without this *get* of the transactions to lower the channel full level, the use of *put* will be blocked when the channel reaches maximum level. This can cause operational problems when the generator of the *put* needs to be nonblocking. For example, a monitor collects words of a packet from an interface and needs to add at every cycle, in a nonblocking manner, the collected data into the scoreboard channel (otherwise, new words from the same packet will be missed). The scoreboard is not ready to process that data until the end of packet signal is received, and many packets words are stored into the channel. The packet size can vary, and fixing the channel level to a fixed value may not guarantee that the maximum will not be reached. Thus, the use of the nonblocking *sneak* to add items into the channel solves that nonblocking need.

The prototype for the *peek* method is:

```
task peek(output class_name obj,input int offset = 0);
```

*vmm_channel::peek g*ets a reference to the next transaction descriptor that will be retrieved from the channel at the specified offset without actually retrieving it. If the channel is empty, the function will block until a transaction descriptor is available to be retrieved. Essentially, a *peek* allows a functional transactor to see the next transaction in order to make decisions about the current transaction in progress.

2.3.3 Channel allocation

An instance of a channel is allocated with the predefined function *new* as shown below:

```
function new(string name,              // specified name
             string instance,          // instance name
             int unsigned full = 1,    // full level default to 1
             int unsigned empty = 0,   // empty level default to 0
             bit fill_as_bytes = 0);   // level is transaction
```

"The *new* function creates a new instance of a channel with the specified name, instance name and full and empty levels. If the *fill_as_bytes* argument is TRUE (i.e., non-zero) the full and empty levels and the fill level of the channel are interpreted as the number of bytes in the channel as computed by the sum of *vmm_data::byte_size()* of all transaction descriptors in the channel, not the number of objects in the channel. If the value is FALSE (i.e., zero), the full and empty levels and the fill level of the channel are interpreted as the number of transaction descriptors in the channel. It is illegal to configure a channel with a full level lower than the empty level."

Note that the *fill_as_bytes* ==1 is useful for Transaction-Level (TL) modeling.[17] For example, consider the case of a video CODEC model with data buffering capability. The size of that buffer is likely limited, but in a TL model, you would not want to model the actual RAM used to implement that buffer. A VMM channel is just perfect for this application, but the amount of memory a packet (or video frame) takes depends on its size. By default, a VMM channel simply counts the number of transaction is has, not how big they are. With this parameter ON, the channel is "filled" as a function of the size of what it contains. So it can have lots of small packets or a few large video frames. The *fill_as_bytes* argument changes how the FULL level of the channel is computed. It is useful in a transaction-level model of a bandwidth-limited transport medium or transfer function. However, for most applications, the default value of *fill_as_bytes*==0 will work.

An example of the *new* is in the FIFO environment (see *fifo_env.sv*) and is shown below:

```
// channel instantiation
  Fifo_xactn_channel fifo_channel_0;
function void build();
 ..
  this.fifo_ channel_0 = new("fifo_chan","0"); // allocation
```

[17] http://verificationguild.com/modules.php?name=Forums&file=viewtopic&p=5307#5307

2.3.4 Channel reconfiguration

The full level of a channel can be dynamically reconfigured with the *reconfigure* function. The prototype of the *reconfigure* function is:

```
function void reconfigure(
   int full = -1, // full level,
   int empty = -1,  // empty level
   logic fill_as_bytes = 1'bx
   );[18]
```

For example, to reconfigure the maximum level (referred to as the full level) of the FIFO channel to 3, you could write:

```
this.fifo_channel_0.reconfigure(.full(3));
```

[18] VMM Appendix A, *vmm_channel reconfigure()*. Reconfiguration may cause threads currently blocked on a *vmm_channel::put()* call to unblock.

2.4 FILE STRUCTURE

Table 2.5 demonstrates the file Structure in the downloaded code for this chapter, and the purpose of each file. Figure 2.5 is a graphical representation of the relationship between the files.

Table 2.5. File Structure and Functions

File	Function	Used by
fifo_pkg.sv	Defines types and parameters	ALL
fifo_if.sv	Defines the FIFO interface	RTL, property models, and by program, testbench, transaction and transactors
fifo_csr_if.sv	Defines the FIFO configuration interface	RTL, property models, and by environment, and possibly transactors
fifo_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (Fifo_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator, monitor transactor for creation of transaction from observed values on bus interface.
fifo_rtl.sv	Represents the FIFO RTL DUT.	Top level
fifo_props.sv	Defines the properties for assertions	Top level for bind

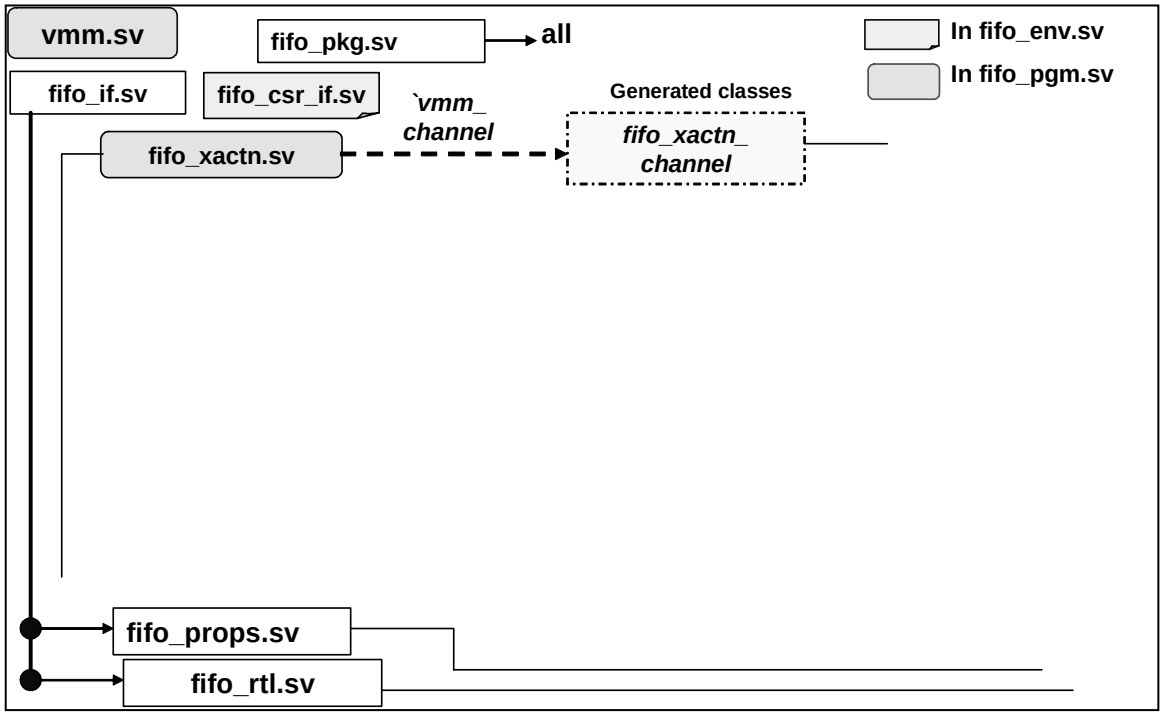


Figure 2.5 File Structure and Relationships

Chapter 2 Questions and LAB

Q1. Why are transactions specified in a class instead of a structure within a transactor?

Q2. Why can't transactions specified in a class be used without the need of a channel?

Q3. How do you build a custom channel?

Q4. If I extend my transaction class (e.g., class Fifo2_xactn extends Fifo_xactn) do I also need to define and use a new channel?

<u>Lab02</u>. Build a Transaction Class and Channel Class for a loadable counter.
See instructions in subdirectory lab/lab02/todo/readme.txt.

Figure Lab02 represents the model for a loadable counter.

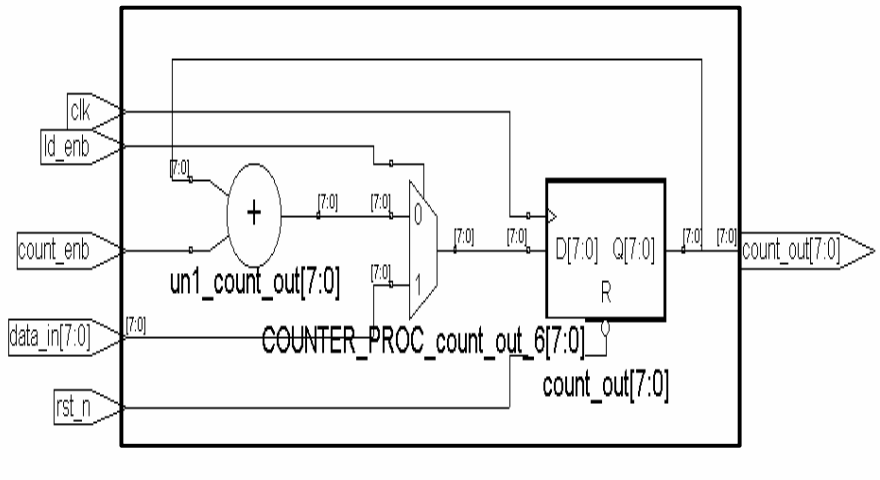


Figure Lab02 Loadable Counter for Labs