# Network System Verification with VERA

Senthil Krishnamoorthy
Gorav Arora
Rajesh Guravannavar

Redwave Networks Inc.
3000 Orchard Parkway, Suite 200
San Jose, CA 94134

skrishnamoorthy@redwavenet.com
garora@redwavenet.com
rguravannavar@redwavenet.com

**ABSTRACT**

Redwave Networks is building the next generation metro network systems. This paper presents the verification environment built using VERA to test multiple ASIC and FPGA designs in stand-alone and subsystem configurations. The environment is specifically architected for code and test portability across multiple sub-system builds. This is achieved by exporting the control paths of all objects encapsulating the various interfaces and data representations into a "knob-base". Tests are written using this high level of abstraction, and control the values for different system parameters in directed-random testing. Moreover, tests can be auto-generated and guided using DUT output feedback and stimulus characterization constraints. Augmenting this methodology with the inherent object-oriented features of VERA has led to a flexible test environment that could simulate a variety of network conditions and achieve test coverage for all chips.

## 1.0 Introduction

In this paper we present a verification flow for a multi-service metro network system. The objective is to test individual chips in standalone and multiple subsystem configurations. The subsystems include custom designed and vendor supplied chips. The verification task involves developing the tools, such as individual drivers and checkers, generating test benches and writing tests. We divide these tasks into two categories. The first task is to build a usable and comprehensive verification environment, and the second is to write and execute tests. Furthermore, the tests and the run-time environment needs to be accessible to a wider audience who are not intimately aware of the implementation details of the test environment.

The verification environment has to be functional quickly, within weeks after chip and system architecture specification. The environment is required to be efficient in terms of run-time and memory utilization. Additionally, each chip design in the verification flow has unique constraints and thus it is often simpler to view the verification of each chip independently. We argue here that it is just as important to establish a framework for the entire verification effort. An encompassing functional checker for each chip is a key element of this flow. The checker models the expected behavior of a chip at specific architectural/micro-architectural boundaries but need not be cycle accurate. The checker understands the internal states of the chip (such as register settings) and models such changes at the specific boundaries. The tests only control the stimulus to the chip/sub-system. In addition to functional checking, several protocol checkers are added that model the interfaces at chip boundaries and other important micro-architectural boundaries. The protocol checkers fill the requirements for checking timing relationships. The software architecture for the environment is required to permit partitioning and code sharing between the developers and consists of a simple build process consisting of makefile hierarchies to create the various test benches. The run-time environment also includes a post-simulation step to enable parsing of patterns from the run-time output of the simulation.

In addition to building the environment to run tests, a large number of random directed tests are needed for several test benches. Tests should have the level of abstraction required to specify system or subsystem level flows, while enabling specification of micro-architecture (level) test corners. One of the tangible benefits of making tests accessible to designers is to reduce the verification effort by enabling designers to specify interesting stimulus for micro-architectural corners. The tests should be useful across multiple test benches, while they model a sequence of events for a specific test bench. Test abstraction should enable performance testing as well as functional testing. A run-time environment to launch and monitor large volume runs is also required to provide test automation.

We use VERA to develop the verification environment. The object-oriented flow of VERA simplifies the construction of all classes from the ground up. The random constraints in VERA provide powerful tools to create directed random tests. This helps in applying useful first pass tests on the DUT. Code sharing is primarily achieved through base classes for drivers, snoopers, checkers, run-time statistics and aggregation.

Tests are written using ASCII files (knob files) that provide control flow for the various classes. Each object in the verification environment makes queries to a knob class. The knob class provides a centralized control flow between objects from the knob files. Often scripts can generate knob files during volume runs. A centralized control flow makes it easier to write run-time specification for control parameters across the system. The run-time environment supports launching and monitoring jobs across a simulation farm while accumulating statistics over the duration of the verification effort.

Knob files are an incomplete specification of test parameters. A test file written for any chip is useful to test subsystems with that chip and vice-versa. System level parameters (knobs), such as bandwidth, QoS parameters and SONET states can conflict with lower level knobs such as inter packet gap, IP header parameters and specific attributes of a SONET frame. We resolve these conflicts by a simple ordering structure wherein a later specification will override any earlier specification for a knob. This flow of knob values is asymmetrical in the current implementation. Having a mechanism to resolve conflicts permits us to overlap different knobs designed to test corner cases. Multiple knob definitions can also be modified with specified simulation time between knob transitions. In order to support dynamic control over the test we need a statistics block. This centralized block holds information on all the interesting events on a chip during a test, and supports queries to dynamically alter test flow.

In section 2, we describe the knobs and the implementation of knobs in VERA. Section 3 describes the complete verification environment. Section 4 presents the success we had with this flow. In section 5, we describe plans to enhance the knob structures. We have included some simple examples in the Appendix to illustrate the implementation of the environment and the knobs that control the environment.

## 2.0 Composing tests without code

In a verification environment, the behavior of a DUT is validated through a test (random or directed). This test applies a set of stimuli to a particular configuration of the DUT. The response of the DUT is subsequently measured and compared against expected behavior.

In common practice, the control parameters for the stimulus and DUT configuration for different tests are expressed as different code structures. To broaden the breadth of testing, these code structures are augmented to include the randomization of the control parameters. Consequently, additional code structures are required to narrow the scope of the randomization to generate a meaningful class of tests.

### 2.1 Knobs

In the presented verification environment, the control parameters of a test are abstracted to a *knob-base*, with each parameter represented by a *knob*. A knob is an abstract construct whose properties are controlled externally, and provides the value of a parameter in the test environment that is required to vary in the life cycle of testing the DUT. The value provided by the knob is directly dependent on the properties of the knob itself. Knobs may control the value of a single control parameter or a number of control parameters in the environment. A knob may not only control values of parameters, but may also be used to control program flow.

A knob may also control the value of one or more knobs. This hierarchical organization allows tests to be written with a very high level of abstraction. Tests are written in an ASCII file, referred to as the *knob file*, which constitutes of the values for all knobs that are required by the test and are described in section 2.2. Each test need only specify the values of certain knobs, while sharing a common, debugged code base. The extent to what a knob may control is only limited by the scope of the knob-base implementation in the test suite and the proliferation of knobs in the architecture of the code structures.

This methodology was fostered by the object-oriented architecture and constraint-driven random stimulus generation of VERA [1]. Extensive use of encapsulation and inheritance in the verification environment architecture allowed the exporting of all control paths for objects representing the various interfaces and data representations in the network system under test. All test platforms share two classes that encapsulate all the necessary functionality and behavior of knobs. The classes are described in the following sections, while an example detailing the specifics of the implementation is shown in the Appendix.

### *2.1.1 CKnob*

The class CKnob encapsulates the behavior and state of a single knob. The class stores the value of the knob in a random integer whose value is controlled using multiple constraint blocks. Each constraint block defines the properties of the knob, i.e., whether the knob has certain fixed values, or a value from multiple ranges. The class provides external methods to set and select the constraints and read the value of the knob. Every read of the knob can provide a new random value or the current value set by the last randomization. Each knob also maintains a multi-level shadow map, which allows knob constraints to be changed and restored dynamically.

### *2.1.2 CKnobTable*

The class CKnobTable provides the external interface of the knobs to the actual tests. It comprises of a local associative array of CKnob and provides necessary methods to manipulate each of them individually. This class is also responsible for parsing the knob files themselves. An example of such knob file is shown in Figure 2.1.

During parsing, each knob name, such as “PACKET_SIZE”, is used to uniquely identify a particular instance of a knob. Depending upon the definition of the values of the knobs in the file, a constraint block is chosen, and the values of the constraint are set. This is repeated for each knob defined in the knob file.

Knobs may be shared across multiple instances of the same object. Each instance of an object constructs the name of the knob it accesses by prefixing an identification string passed to its constructor. This allows some instances of the same object to share knobs, while others use unique knob groups.

The CKnobTable class provides a more abstract interface to CKnob. It provides a generic task to access the CKnob instances using the actual string name of the knob. It further extends the CKnob by providing methods to manipulate each knob’s shadow maps and constraint blocks.

## 2.2 Test Composition

At the start of each test, all knobs are initialized to default values using a separate knob file. Subsequently, a knob file particular to the test is parsed and the necessary knobs are re-initialized. This is performed before the instantiation of any data or control objects. The values of the knobs can be changed dynamically during the simulation, even for interim periods based on certain events.

Events that trigger the change of knob values can be time-based events, such as elapsed simulation time, or measured DUT environment conditions. Events can also be based on a per stimulus basis, where the generation of a certain stimulus causes another set of stimuli to be generated. Thus, a knob can change its subsequent value based on its current value. This can be used to generate a sequence of stimuli, such as a stream of a sequence of packets. The exact event and consequent changes are solely dependent upon the exact code implementation by the programmer.

***Figure 2.1:*** An example of a knob file

```
START_KNOB_DEFINITION
KNOB    PACKET_SIZE
        64      128     90
        256     512     10

KNOB    NUM_PORTS_TO_TARGET
        1       8       70
        9       16      30

KNOB    PORTS_TO_TARGET
        0       15      100

KNOB    DIST_OF_PORTS_TO_TARGET
        1       95
        2       5

KNOB    HEAVY_LOAD_ON_PORT
        TRUE            25
        FALSE           75
...

END_KNOB_DEFINITION
```

An example of a knob file (or test) is shown in Figure 2.1. It illustrates the flexibility and granularity of the knobs that have been architected into the system. Suppose the above knob file is used to generate tests for a 16-port switch. The knob file defines with knob PACKET_SIZE that on average, 90% of the packets would be between 64-128 bytes, while 256-512 bytes 10% of the time. The knob NUM_PORTS_TO_TARGET defines that most tests would more likely target 8 or less ports of the switch. However, any of the physical ports is equally likely to be used in a test (PORTS_TO_TARGET). The DIST_OF_PORTS_TO_TARGET dictates the distribution of weights for each of the physical ports, i.e. each 95% of the physical ports chosen would be equally likely to be targeted, however, 5% of the ports would be targeted twice as much. Finally, through the course of the test, a pre-defined knob HEAVY_LOAD_ON_PORT in turn sets the other knobs such that only large packets are generated to a single port for a pre-defined period of time. A more complete example is shown in the Appendix. Evidently, the definition, interpretation and use of the knobs are dependent upon the test environment.

## 2.3 The Silver Bullet

The advantage of adopting the aforementioned methodology becomes apparent during the life cycle of system verification. Since the mature code base is shared among all the system and sub-system tests, only different knob files need to be generated for tests targeting chip, sub-system and system levels. The stimulus generation code structure relies completely on the knob files to provide all constraints for the random parameters

used in test, and thus a single knob file can yield a nearly exhaustive combination of stimulus and DUT configurations for a particular test type.

Consequently, new tests can be rapidly generated, shortening the verification phase cycle. Since all tests share a common, debugged code base, the overhead associated in maturing a network system verification environment from the early chip level testing to eventual system level testing is greatly reduced. Moreover, since composing of tests is programming language independent, it enables a wider audience (such as the designers) to write or modify tests.

The environment "auto-generated" tests using the hierarchical knobs. By simply specifying the values of certain high level knobs (such as NUM_PORTS_TO_TARGET), the test would choose a grouping of ports to target, and subsequently each of their configuration properties and stimulus generation. During simulation, tests can be guided using DUT output feedback and stimulus characterization constraints to dynamically change the knob values. This yields a very powerful tool, capable of maximizing the random stimulus generation inherent in VERA.

However, such an approach does have a few drawbacks. The methodology increases the initial development time requirement. Extra measures need to be taken to ensure knob based control extends throughout the verification environment architecture. The code complexity also increases since knob data structures are to be maintained and used. Thus, additional time is required to write the test environment.

## 3.0 Implementation

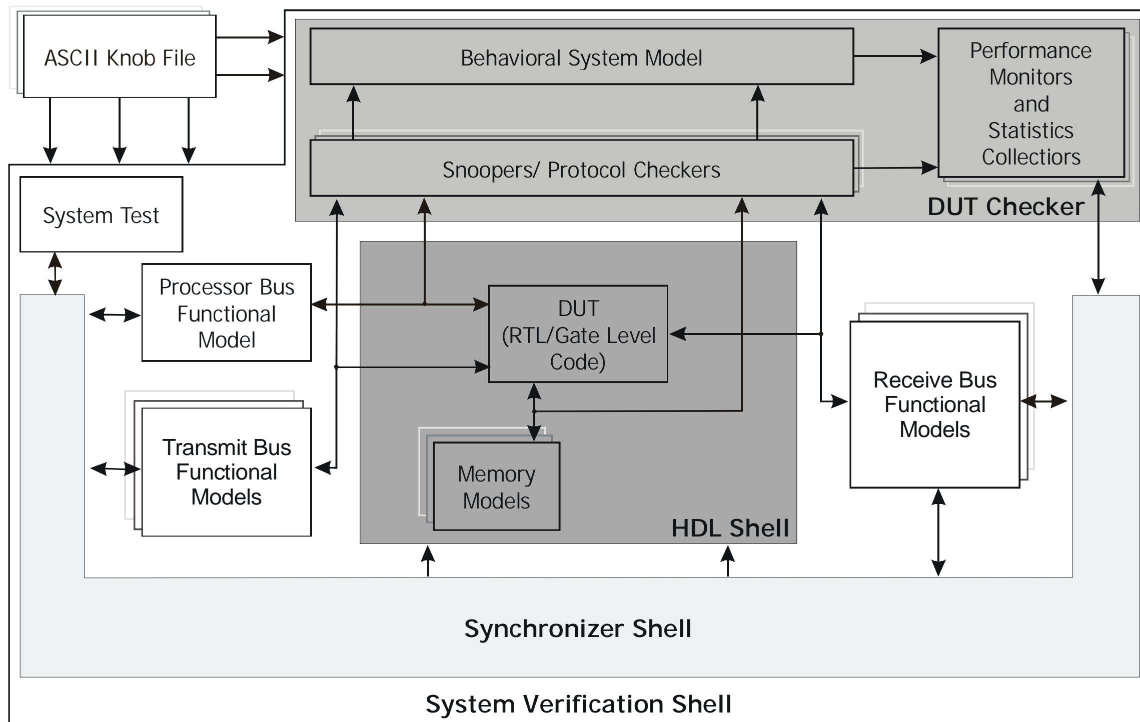


***Figure 3.1***: Proposed architecture for knob based verification environment for network systems

The proposed knob based verification environment architecture is shown in Figure 3.1. Each of the functional blocks shown in Figure 3.1 is comprised of one or more classes that were designed using object-oriented analysis and design techniques [2]. All class definitions and interfaces were constructed to maximize their usability across multiple chip/sub-system environments of the verification infrastructure. The strict enforcement of data encapsulation enabled a parallel development effort of the verification environment. The use of inheritance (e.g. to construct different network layer/protocol packets), and aggregation allowed the smaller chip level environments to be effortlessly integrated into a system/sub-system level verification environments. Each of the components include a common knob interface class CKnobTable (described in section 2.1.2) allowing them to access the central configuration parameters.

### 3.1 Synchronizer

The Synchronizer class is a top-level shell for a particular chip or sub-system configuration which instantiates all the objects and binds them to the HDL shell ports. It also maintains data structures such as packet buffers and performs synchronization between objects in separate clock domains, concurrent processes of different objects and architectural boundaries. It performs this synchronization through lists and semaphores, and provides error trapping and top-level simulation start and stop routines. For example, the snoopers and the DUT behavioral model often lie in different clock domains. The Synchronizer synchronizes the events generated by each of these components using local lists and semaphores.

The Synchronizer also performs memory management to enforce fairness across multiple objects, as well as ensure that memory resources are used at a reasonable level throughout the duration of the tests. Memory requests that can be queued are tailored to fork subsequent processes that cannot wait for memory allocation. For example, a packet generator may request memory for certain number of packets and subsequently trigger an event upon filling a buffer. The event is received by the Synchronizer, which would consequently trigger the send routine in the bus functional model. A new memory request to the Synchronizer from a packet generator is queued, until the bus functional model signals successful transmission of all the packets in the buffer. This limits the number of outstanding packets in the system.

### 3.2 Bus functional models

Bus functional models (or drivers) were developed to emulate boundary interfaces of the third-party chips, such as processors, and custom chips under development. They comprehensively encompass low-level protocol details to conduct the required transactions on the interface they encapsulate, while providing an abstract interface. This enables other components to communicate with the model using high-level information (e.g. packets), leaving the “per-cycle” state of the communication bus to the model. The exact behavior of the model is controlled through knob parameters, allowing complete protocol level coverage.

### 3.3 Protocol Checker / Snoopers

Separate protocol checkers and snoopers were developed, to allow the VERA bus functional models to be replaced with HDL models for the surrounding chips as the system matured to sub-system configurations. Protocol checkers promiscuously view all transactions on the communication bus interfaces and enforce timing relations among the signals. They also model certain properties of the interface device such as internal buffers to check under-run, over-run, etc. conditions.

Snoopers record all transactions on an interface of a DUT. They do not enforce any level of checking, however, they provide the transaction information to other components such as DUT checkers and behavior models. They convert the low level bus transaction into a more usable, higher-level abstract forms required by the components (such as packets). The snoopers shared some of code base as the bus-functional models to reduce the development time and maintenance.

Development of the drivers/snoopers and protocol checkers was divided amongst developers of complex bus protocols to minimize interpretation errors.

### 3.4 DUT Behavioral Model

The DUT behavioral model mimics all the state transitions of a DUT given the identical stimulation. All stimulus applied to the DUT is captured by the snoopers and applied to the model. The models are generally not cycle accurate, since packets are the basic processing block within most DUTs. Thus, the behavioral models are packet accurate. However, in some cases to precisely validate the DUT response, it was necessary to perform cycle accurate calculations (such as traffic policing and packet dropping).

### 3.5 Checkers

Checkers compare actual and expected DUT behavior. The actual behavior is determined by the capture of DUT output by the snoopers, while the DUT behavioral model determines the expected behavior. The checkers also receive information from other blocks performing statistics collection, performance monitoring to enforce certain higher-level system requirements. The checker enforces packet ordering, sequencing and verifies the correctness of the packet switching fabric.

### 3.6 System Test

The primary responsibility of the system test is to configure the system and stimulus generation. It instantiates and initializes the Synchronizer and global knob-base from a particular knob file. All subsequent object initializations use the configured knob-base to initialize their state. The system test may re-configure the knobs based on knobs contained in the knob file. Once all components have been initialized, the test triggers the start of simulation. During the execution of the test, the system test may also use dynamic events to reconfigure the knobs. The system test may also perform post-processing of simulation data such as bandwidth statistics and summary report generation.

## 4.0 Performance

The success criteria for a verification flow is determined by its ability to provide test coverage, and isolate the bugs in the various DUT. We achieved very high line coverage for all the chips under test using CoverMeter. We should add that most of our regressions run without CoverMeter, and many of the random knob sequences do not repeat. Our sub-systems tests were useful to identify architecture and configuration issues, and board level connectivity errors. Many of our chips and boards are currently in the lab under system test. The verification effort for 5 chips (ranging from hundreds of thousands of gates to multi-million gates) and sub-systems was completed in less than a year, with small team of verification engineers. Modular design and easy to use methodology should enable faster turn around as we continue to build our next generation system.

## 5.0 Future Work

In this section, we list some features that will enhance our implementation of knobs. In the current implementation, we require special knobs to target every interesting sequence of events. This increases the number of knobs associated with a sub-system. We also have a specific relationship in the test to generate other knobs, when one or more values change during the test. We address these issues by adding two key features, *sequencing* and *dependency lists*. For sequencing, we maintain a count of the queries made on each knob. A specific sequence number of the query triggers changes to one or more knobs. With dependency lists, we plan to support the syntax to specify the relationships between knobs explicitly in the knob file.

## 6.0 References

[1] F. Haque, J. Michelson and K. Khan, *The Art of Verification.* Fremont: Verification Central, 2001.

[2] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and theUnified Process*. New York: Prentice Hall, 2001.

## 7.0 Acknowledgements

We would like to extend our gratitude to Mehdi Mohtashemi, Technical Director at Synopsys for understanding our initial requirements and for his valuable suggestions. We would also like to thank James Caviness for playing a key role in the initial concept of knobs.

# 8.0 Appendix

## 8.0.1 Pseudo-code example for class CKnob

An example pseudo-implementation of the CKnob class is given in Figure 8.1.

```
////////////////////////////////////////////////////////////////
// Class CKnob: This encapsulates the behavior and properties of the Knob abstract construct
class CKnob {

        local rand integer m_nValue;            //The current value of the knob.
        local integer m_nUpperBound[MAX_BINS], m_nLowerBound[MAX_BINS];  //The valid ranges a knob can have.
        local integer m_nWeight[MAX_BINS];     //The weights for each of the ranges

        //The shadow values
        local integer m_nShadow_UpperBound[MAX_BINS], m_nShadow_LowerBound[MAX_BINS];
        local integer m_nShadow_Weight[MAX_BINS];

        //Constraints for a knob with set values.
        constraint SetValues {
                m_nValue dist {
                        m_nLowerBound[0] := m_nWeight[0];
                        m_nLowerBound[1] := m_nWeight[1];
                        ...
                };
        }

        //Constraints for a knob that can have values from ranges
        constraint RangeValues  {
                m_nValue dist {
                        m_nLowerBound[0]:m_nUpperBound[0] :/ m_nWeight[0];
                        m_nLowerBound[1]:m_nUpperBound[1] :/ m_nWeight[1];
                        ...
                };
        }

        task new(integer constraint_type) ;
        local task ZeroAllWeights();                        //clear the knob state
        function integer GetKnobValue(bit NewRandomVal);    //Return the current, or new value of the knob
        task SetKnobValue(index, Value, Weight);            //Set a value for a knob
        task SaveKnobState();                               //Save the state of the knob to the shadow.
        task RestoreKnobState();                            //Restore the state from the shadow
}

task CKnob::new(integer ConstraintType) {
        ZeroAllWeights();
        void = constraint_mode(OFF, "SetValues");
        void = constraint_mode(OFF, "RangeValues");

        //select which constraints should be active
        if (ConstraintType == SET_VALUES)
                void = constraint_mode(ON, "SetValues") ;
        else if (ConstraintType == RANGE_VALUES)
                void = constraint_mode(ON, "RangeValues");

        ASSERT(this.randomize() != RAND_FAILURE); //set the initial value to be a valid one.
}

function integer CKnob::GetKnobValue(bit NewRandomVal) {
        if (NewRandomVal) {
                ASSERT(this.randomize() != RAND_FAILURE);
                GetKnobValue = m_nValue;
        } else
                GetKnobValue = m_nValue;
}
```

***Figure 8.1 Pseudo-code example for class CKnob***

### 8.0.2 Class definition of class CKnobTable

The class definition for class CKnobTable is shown in Figure 8.2. The class manages a local array of CKnob objects, and simplifies their use.

```
//////////////////////////////////////////////////////////////////
// Class CKnobTable: This class provides an external interface to manage a large number of knobs.
class CknobTable {
        //The array of knobs the class manages
        local CKnob m_Knob[];

        //Lookup functions
        local function integer KnobName_TO_Index(string KnobName);
        local function string KnobIndex_TO_Name(integer KnobIndex);

        //Read and parse a knob file
        task OpenKnobFile(string FileName);

        //Get the value of knob as specified by KnobName
        function integer GetKnobValue(string KnobName, bit NewRandomVal=TRUE);

        //Set the values for a knob which can only have a set number of values. Save the current properties.
        task SetKnobValue(string KnobName, integer Values[], integer Weight[]);

        //Set the ranges from which a knob can choose a value from.  Save the current properties.
        task SetKnobRange(string KnobName, integer LowerBound[], integer UpperBound[], integer Weight[]);

        //Restore the Knob from the save properties.
        task RestoreKnob(string KnobName);
}
```

***Figure 8.2 Class definition for class CKnobTable***

### 8.0.3 Bus Functional Model Example

A POS-PHY Level 3 bus functional model example is shown in Figure 8.3.

```
//////////////////////////////////////////////////////////////////
// Class CPosPhyDriver: This class encapsulates the low-level protocol details of a POS-PHY Level 3 bus.
class CPosPhyDriver {
        local m_PL3TXBus, m_PL3RXBus;
        local VeraList_CEthernetPacket m_RXPacket[MAX_RX_PACKETS];
        local string m_ID;
        public event m_eRXComplete;
        …
        local task Receive ();                              //Receive a packet
        public task new (string object_id, PL3TXPort  new_pl3_tx_bus, PL3RXPort new_pl3_rx_bus) {
                m_PL3TXBus = new_pl3_tx_bus;
                m_PL3RXBus = new_pl3_rx_bus;
        }
        public task InitBus();
        public task Transmit (CEthernetPacket packet);      //Transmit a packet
        public task GetRXPkt(var CEthernetPacket packet);      //Return a received packet
        …
}

task CPosPhyDriver::Transmit(CEthernetPacket packet) {
        for (i=0; i < packet.GetSizeInWords(); i++) {
                @ (posedge m_PL3BUS.$TCLK);
                NumStallCycles = KnobTable.GetKnobVal({m_ID, "PL3_TX_STALL"});

                if (NumStallCycles > 0)
                        TXStall (NumStallCycles);  //function returns after stalling for NumStallCycles

                ...        //Drive out single cycle of data
        }
```

***Figure 8.3 Example pseudo-code for a POS-PHY Level 3***

### 8.0.4 Packet Class Example

Sample pseudo-code is shown in figure 8.4 for a class that encapsulates Ethernet packet information. The class is capable of generating Ethernet packets that comply with the configuration specified through knobs.

```
////////////////////////////////////////////////////////////////////
// Class CEthernetPacket: A packet class that is the basic processing block of network systems
class CEthernetPacket extends CBasePacket{
        …
        public task Create(); //Create a new packet matching the configuration parameters (knobs)
}

CEthernetPacket::Create() {
        ...
        m_Size = KnobTable.GetKnobVal("PKT_SIZE");
        m_MACSrcAddr  = KnobTable.GetKnobVal("MAC_SRC_ADDR");
        m_MACDestAddr = KnobTable.GetKnobVal("MAC_DEST_ADDR");
        m_DestIPAddr = KnobTable.GetKnobVal("DEST_IP_ADDR");
        m_NumMPLSTags = KnobTable.GetKnobVal("NUM_MPLS_TAGS");
        ...
}
```

***Figure 8.4 Pseudo-code for a Ethernet Packet class***

### 8.0.5 Synchronizer class example

Figure 8.5a and 8.5b show example pseudo-code for a synchronizer. The presented synchronizer interfaces between a System test of a DUT with POS-PHY Level 3 input and outputs and the bus-functional models.

```
////////////////////////////////////////////////////////////////////
// Class CSynchronizer:  This class instantiates all objects and provides synchronization amongst them
class CSynchronizer {
        local CPosPhyDriver TXPL3Driver[MAX_NUM_PORTS];      //POS-PHY bus drivers for TX and RX
        local CPosPhyDriver RXPL3Driver[MAX_NUM_PORTS];
        local VeraList_CEthernetPacket TXPacket[];           //Local list to enable synchronization across
        local VeraList_CEthernetPacket RXPacket[];           //objects.
        …
        local task DrivePacket(int port_num); //Provide synchronization between POS-PHY bus and external objects
                                              //Call low level driver to transmit packet
        ...
        task new();
        task SendPacket(int port_num); //Send a packet on given port number
        task ReceivePacket(int port_num, var CEthernetPacket packet);  //Receive packet from a port.
}

CSynchrnoizer::new() {
        ...
        for (i=0; i<MAX_NUM_PORTS; i++) {
                TXPL3Driver[i].Init();
                fork
                        DrivePacket(0);
                join none
        }
}

CSynchorizer::SendPacket(int port_num) {
        ...
        semaphore_put(m_semSendPacket[port_num],1);
}
```

***Figure 8.5a Example pseudo-code for a Synchronizer***

```
CSynchronizer::DrivePacket(int port_num) {
		CEthernetPacket Packet;

		while (!SimDone())  {
			semaphore_get (m_semSendPacket[port_num],1);  //Synchronize with calling component
			...							//Other Synchronization events
			...							//Stream generation events
			Packet.Create();					//Create a new packet
			TXPL3Driver[port_num].Transmit(Packet);//Send Packet on bus
			...							//Synchronization events
		}
}
```

***Figure 8.5b Example pseudo-code for a Synchronizer (Continued)***

### 8.0.5 Synchronizer class example

Figure 8.6 shows example pseudo-code a knob based system test.

```
////////////////////////////////////////////////////////////////////////
// A simple example to demonstrate the use of knobs.
program KnobExample {
		CKnobTable KnobTable;
		CSynchronizer Synchronizer;
		integer Ports[], Weight[];
		...
		//Initialize the Knob Table
		KnobTable = new();
		KnobTable.ReadKnobFile("Figure2.1.knb");

		//select how many ports are going to be active in the test
		NumPortsToSend = KnobTable.GetKnobVal("NUM_PKTS_TO_TARGET");

		//select which ports are going to be targeted, and their weight
		for (i=0; i < NumPortsToSend; i++) {
			Ports[i] = KnobTable.GetKnobVal("PORTS_TO_TARGET");
			Weight[i] = KnobTable.GetKnobVal("DIST_OF_PORTS_TO_TARGET");
		}

		//Re-write the knob table for the test.
		KnobTable.SetKnob("PORTS_TO_TARGET", Ports, Weight);
		Synchronizer.Init();  //ensure that all objects are configured using the knobs
		Synchronizier.Start();  //Start the concurrent processes of the environment
		while (!SimDone()) {
			//Decision to increase the load on a port
			if (KnobTable.GetKnobVal("HEAVY_LOAD_ON_PORT")) {
				//Re-write the necessary tables to accomplish test condition
				KnobTable.SetKnob("PORTS_TO_TARGET", \
							KnobTable.GetKnobVal("PORTS_TO_TARGET"), 100);
				KnobTable.SetKnob("PACKET_SIZE", 1536, 100);

				//Generate a burst of traffic
				for (j=0; j < 10; j++)  {
					Synchronizer.SendPacket(KnobTable.GetKnobVal(“PORTS_TO_TARGET));
				}

				//Restore the Knobs to the previous state
				KnobTable.RestoreKnob("PORTS_TO_TARGET");
				KnobTable.RestoreKnob("PACKET_SIZE");
			}
			...		//other events to change knob values
			else
				Synchronizer.SendPacket();    //Generate traffic
		}
		Synchronizer.Stop();  //stop all processing
		...		//post-processing functions
}
```

***Figure 8.6 Example code for program using knobs***