Verification of a Transport Offload Engine

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI'I IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

DECEMBER 2002

By

Abhay Raut

Thesis Committee:

Todd R. Reed, Chairperson

Galen Sasaki

Tep Dobry

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to my research advisor, Professor Michael J. S. Smith. These past two and a half years of working under his guidance were a truly invaluable and rewarding experience.

I am extremely grateful to Professor Todd R. Reed who served as a Chairman of my committee and whose care and concern helped me to complete my thesis.

I would also like to express my deepest thanks to Professor Galen Sasaki and Professor Tep Dobry for their advice, readiness to help, kindness and the time they gave to serve on my Committee.

Thanks are due to John Minami, Co-founder and Chief Engineer of iReady Corporation, for his constant encouragement during the entire CPeer development process. Thanks are also due to Robert Lang, Director of Hawaii Design Center, who managed and encouraged this project. I am also thankful to Dan Greenfield, Senior Design Engineer at iReady Corporation, for helping me out me at every stage of the work.

I would like to thank Claudio Talarico for being wonderful co-worker and a friend. We shared lots of happiness and success.

Finally I would like to thank my parents for their constant support, patience and encouragement all the time.

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to my research advisor, Professor Michael J. S. Smith. These past two and a half years of working under his guidance were a truly invaluable and rewarding experience.

I am extremely grateful to Professor Todd R. Reed who served as a Chairman of my committee and whose care and concern helped me to complete my thesis.

I would also like to express my deepest thanks to Professor Galen Sasaki and Professor Tep Dobry for their advice, readiness to help, kindness and the time they gave to serve on my Committee.

Thanks are due to John Minami, Co-founder and Chief Engineer of iReady Corporation, for his constant encouragement during the entire CPeer development process. Thanks are also due to Robert Lang, Director of Hawaii Design Center, who managed and encouraged this project. I am also thankful to Dan Greenfield, Senior Design Engineer at iReady Corporation, for helping me out me at every stage of the work.

I would like to thank Claudio Talarico for being wonderful co-worker and a friend. We shared lots of happiness and success.

Finally I would like to thank my parents for their constant support, patience and encouragement all the time.

Abstract

Technological advances have strong impact on the integrated circuit (IC) complexity and size. Traditional methods used for verification of digital designs have difficulties to handle this growth. It is thus necessary to develop efficient methodologies for designing verification environments. A C++ based verification environment allows reducing the verification effort and contributes to raise the level of abstraction at which testbenches are described. This thesis describes a C++ based verification environment called "CPeer" that was developed to verify the functionality of a Transport Offload Engine (TOE). A TOE is one of the technologies that offload the TCP/IP processing from the host processor to the host bus adapter or network interface card thereby reducing the server-networking bottleneck.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
BFM	Bus Functional Model
CPU	Central Processing Unit
DUT	Design Under Test
FPGA	Field Programmable Gate Array
FTP	File Transfer Protocol
GMII	Gigabit Media Independent Interface
HDL	Hardware Description Language
ICMP	Internet Control Message Protocol
IFG	Inter-Frame Gap
IGMP	Internet Group Management Protocol
IEEE	Institute of Electrical and Electronic engineers
IP	Internet Protocol
IPsec	IP Security Protocol
iSCSI	Internet Small Computer System Interface
LAN	Local Area Network
MAC	Media Access Control
MII	Media Independent Interface
MSS	Maximum Segment Size
NIC	Network Interface Card

NS	Network Stack
OSI	Open Systems Interconnections
PAWS	Protection Against Wrapped Sequence Numbers
PHY	Physical Layer
RFC	Request for Comments
ROM	Read Only Memory
RTL	Register Transfer Level
RTOS	Real Time Operating System
SACK	Selective Acknowledgment
SDRAM	Static Dynamic Random Access Memory
TCP	Transmission Control Protocol
TOE	Transport Offload Engine
UDP	User Datagram Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrate Circuit
VMAC	Virtual Media Access Control

Chapter 1
Introduction

1.1 Project Background

Ethernet has become the most popular local area networks (LAN) technology. It is inexpensive and offers plug-and-play implementation. As networking has grown in popularity, Ethernet network link speeds have also been increasing. The growth of Ethernet from 10 Mbit/s to 10 Gbit/s has surpassed the growth of microprocessor performance in mainstream servers and computers.

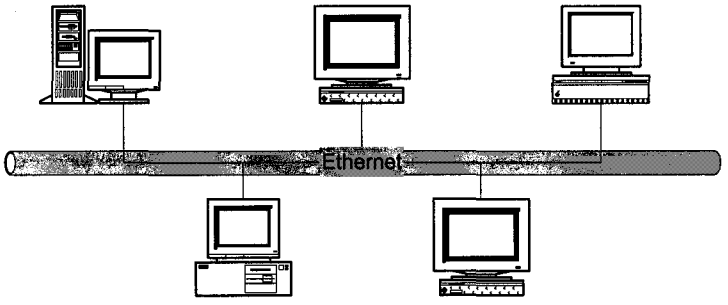


Figure 1.1 An Ethernet LAN

Most computer networks transfer data across a medium at a fixed rate, often faster than the speed in which computers can process individual bits. To solve this problem each computer attached to the network contains special purpose hardware, known as a *Network Interface Card* (NIC), which handles all the details of packet transmission and reception. The NIC operates as an I/O device. The *Central Processing Unit* (CPU) instructs the NIC to begin transmission and then can continue other tasks, while the NIC handles the details of accessing the medium and transmitting bits. The NIC interrupts the

CPU only when the transmission or reception is finished.

The communication between computers on a network is governed by a set of protocols. A protocol is a set of clearly defined rules. By following these rules, the sending node sends a message in a standard format so that the receiving node can interpret and understand it [2]. The following sections describe the OSI reference model and TCP/IP protocol suite.

1.1.1 OSI Reference Model

The OSI (Open Systems Interconnections) reference is an industry standard intended for communications and networking of open systems [2]. "Open" implies the interoperability with any type of communication systems. OSI reference model has seven layers [3][4], which are shown in Figure 1.2

- Layer 1: Physical Layer. It governs the hardware connection and the transmission of bit streams over a communication channel. The transmitting unit of this layer is bit.
- Layer 2: Data Link Layer. It is also called the Network Interface Layer. The transmitting unit in this layer is frame. A frame is a stream of bits that flows across the Ethernet [7]. This layer defines the rules for sending and receiving information from one node to another. It breaks the raw bit streams coming from Physical Layer into data frames, then transmits the frames sequentially and processes the acknowledgement sent back by the receiver. It also provides the error detection, the data link error recovery and modification.
- Layer 3: Network Layer. The transmitting unit in this layer is packet (also called datagram). A packet is the data unit sent to the network interface. This layer makes

routing decisions across the communication network and deals with packet congestion problem.

- Layer 4: Transport Layer. In this layer, the message is decomposed into transmittable packets. It also assigns a sequence number to each packet, so that they can be recomposed at the destination end. It provides end to end communication control. By performing the error detection, it can supervise the validity of transmissions. It also determines what type of service to provide to the session layer, and finally, to the end users.

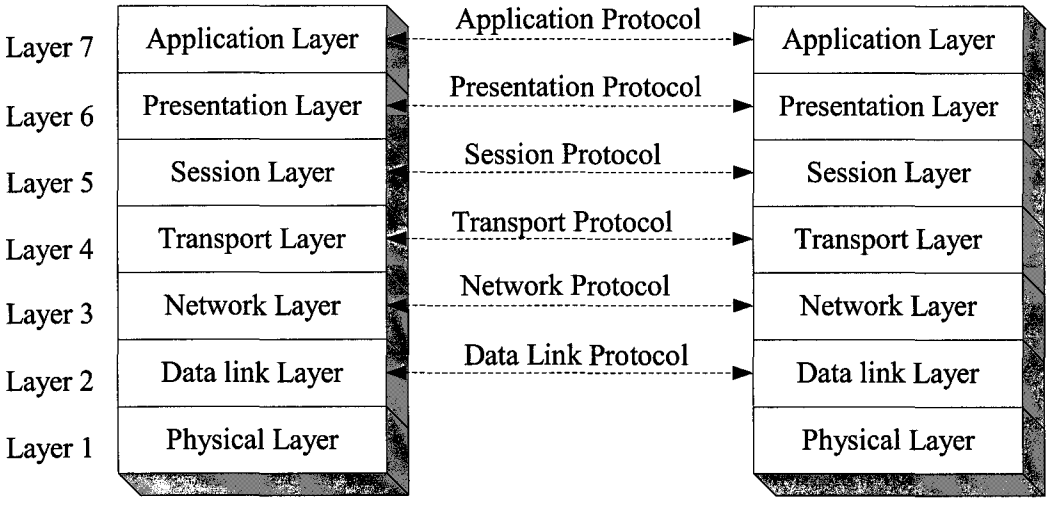


Figure 1.2 OSI Reference Model

- Layer 5: Session Layer. It coordinates the communication (dialog) between different systems. It assumes that a reliable virtual point-to-point connection is made between the two end users until the session is finished.
- Layer 6: Presentation Layer. It provides the transformation of data to standardize the application interface. It also provides some network services, such as encryption, compression, etc.
- Layer 7: Application Layer. It provides a lot of network services to users, such as

electronic mail (E-mail), terminal access, etc. It acts as an "application interface" to the network

Thus OSI reference model is designed as a layered architecture with a series of levels built upon its predecessor.

1.1.2 TCP/IP Protocol Suite

A protocol suite is the combination of different protocols at various layers, so the protocol suite also has layered architecture. The most widely used protocol suite is TCP/IP (Transmission Control Protocol / Internet Protocol). It allows different computers, running on different operating systems, to communicate with each other. TCP/IP makes the Internet possible. [7].

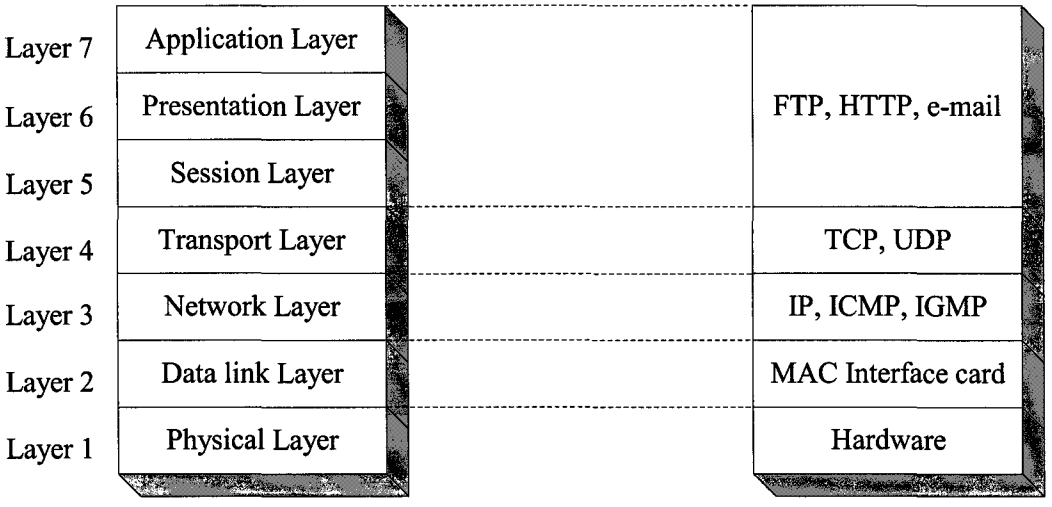


Figure 1.3 OSI Reference Model vs. TCP/IP Protocol Suite

Comparing with the seven-layered OSI protocol, TCP/IP has five layers. The comparison is shown in figure 1.3. Each layer of TCP/IP protocol suite has a different responsibility and can employ different protocols.

- Data Link Layer: It is also called the network interface layer. It handles all the hardware details of physically interfacing with the network. For an Ethernet, the media access control Protocol (MAC Protocol) should be used.
- Network Layer: IP (Internet Protocol), ICMP (Internet Control Message Protocol), IGMP (Internet Group Management Protocol) can be used in this layer.
- Transport Layer: Two vastly different protocols, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) can be used.
- Application Layer: There are many TCP/IP application protocols provided, such as FTP (File Transfer Protocol) and e-mail (electronic mail), etc.

1.1.3 TCP/IP Protocol Processing Bottleneck and the TOE

TCP/IP has matured to become the backbone of the Information superhighway. A fundamental obstacle to improving network performance is that servers are designed for computing rather than for input and output (I/O). The Internet revolution has drastically changed server requirements, and I/O is becoming a major bottleneck in delivering high speed computing [19] The main reason for bottleneck is the underlying TCP/IP protocol stack being processed at a rate less than the network speed.

The processing of TCP/IP over Ethernet is traditionally accomplished by running software on the central processor unit (CPU), of the server. As network connections scale beyond Gigabit Ethernet speeds, the CPU is burdened with large amount of TCP/IP protocol processing. Reassembling out of order packets, resource intensive memory copies, and interrupts put a tremendous load on the host CPU [7] The CPU has to dedicate more processing to handle network traffic than to the applications it is running.

A general rule of thumb is, for every one bit per second of network data processed, one Hertz of CPU processing is required [8]. So for instance, a 20 GHz CPU running at full utilization would be needed to drive a 10 Gbps Ethernet network link. But at this time there is no known processor solution to this problem. So there is a need for dedicated hardware to decode Internet protocols. TCP/IP offload Engine (TOE) is one of the emerging solutions that can reduce the amount of TCP/IP processing handled by microprocessor and server I/O subsystem. Figure 1.4 compares the TOE technology with the traditional TCP/IP software stack.

The basic idea of a TOE is to offload the processing of TCP/IP protocols from the host processor to the hardware on the adapter or in the system. A TOE terminates the Internet connection and leaves at the edge of Internet, not in the middle. Under the umbrella of the TCP/IP protocol stack, there are multiple protocols like TCP, IP, UDP, ICMP and others. Most end-user applications use TCP and IP protocols. A TOE can be implemented with a network processor and firmware, specialized ASIC (Application Specific Integrated Circuits), or a combination of both.

In a discrete implementation, TOE is implemented using microprocessor running a real time operating system (RTOS) and a MAC/PHY. The protocol processing from the host CPU is offloaded to the protocol stack in the RTOS. This implementation can offload the TCP stack and any other protocols that are supported by the embedded stack in RTOS. The advantage of this implementation is the flexibility of the solution and wide availability of the components.

In an ASIC-based implementation, TCP/IP processing is offloaded to performance-optimized hardware. Full offloading executes all phases of the TCP stack in

hardware. With full offload, a TOE relieves the host not only from processing data, but also from connection management tasks. ASIC implementations are customized for TCP/IP protocol offload, offering better performance than discrete implementations. There is general agreement that the advantages of this implementation are performance and scalability, but at the expense of flexibility [8].

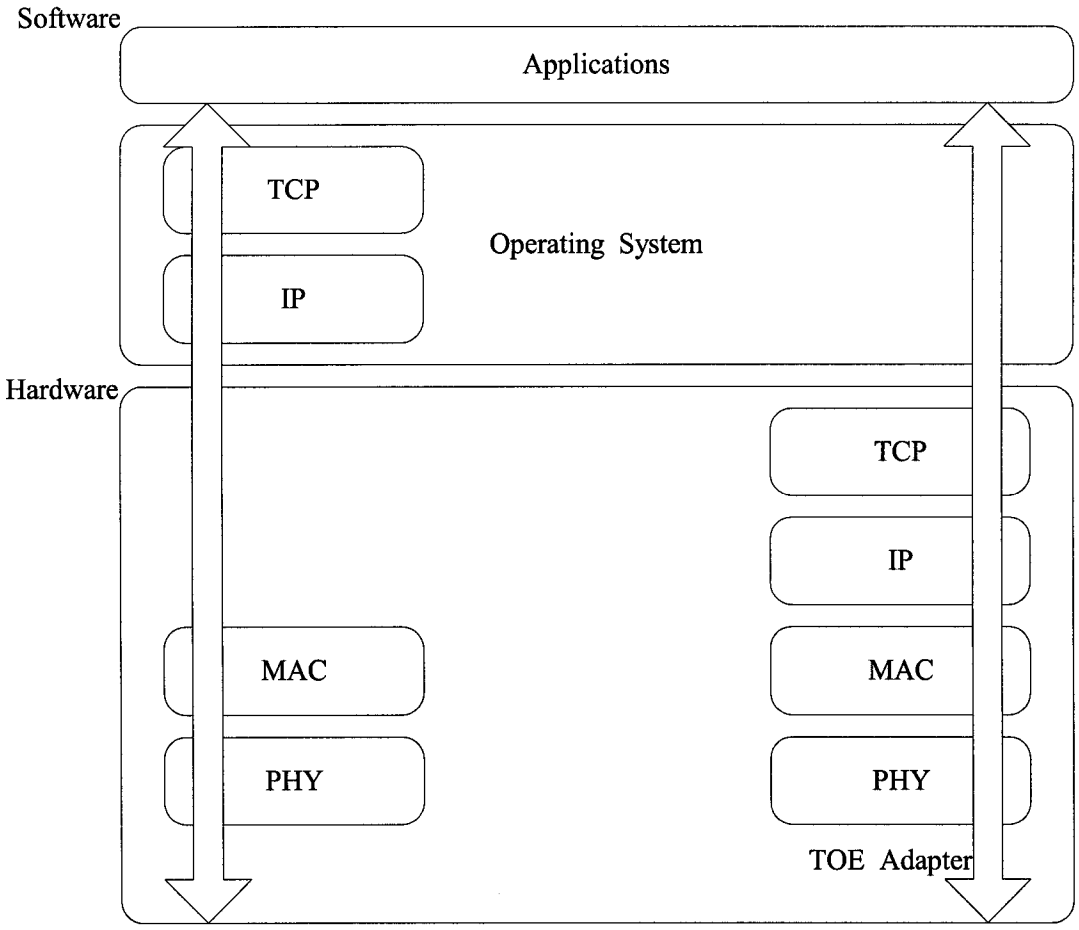


Figure 1.4 Tradtional TCP/IP Stack vs. TCP/IP Offload Engine (TOE)

1.2 Thesis Outline

A critical step to develop a TOE is to verify its design. The thesis project was to develop a functional verification environment called "CPeer". This was part of a large

project at iReady Design Center in Honolulu. The thesis project was under supervision of Dr. Michael Smith, who is the Chief Technology Officer at iReady.

Actually the thesis project was to modify an old version of CPeer. The version had two limitations. First, it was written in the 'C' programming language, and not designed according to the object-oriented paradigm. Second, its capabilities were limited to basic TCP/IP packet generation.

The thesis project involved

- Understanding the old 'C' CPeer code
- Extending CPeer model capabilities by adding the features to handle IP header options processing and TCP header options processing. Support for following header options was added.
    - Maximum Segment Size
    - TCP Window Size Scaling,
    - TCP Timestamp and,
    - Selective Acknowledgment (SACK).

  The TCP window size scaling and Timestamp features were used for Request for Comment (RFC) 1323 [25] verification of the TOE while the SACK feature was used for RFC 2018 [26] verification of the TOE. Enhancements were also done in TCP header and IP header checksum computation functions to support the variable length of the header when the header options are present in the packet. The old checksum functions used to compute the checksum based on the fixed length of 20 bytes.

- Creating an object-oriented verification environment using C++. It involved development of an entire new set of interface classes called *VMAC, MAC*, a core class called *CPeer* and the associated *Connection* classes viz *ICMP, TCP* and *UDP*
- Creating methods for controlling the TOE from the Host interface in the simulation environment. Methods were also added to do concurrent data send and receive operations. These methods were used for generating streams of data from both sides (i.e. CPeer and TOE) for multiple connections and for verifying the correctness of data received.
- Adding logging procedures by incorporating explicit Error/Warning and timeout routines. A generalized logging function, which captured *Information*, *Warning* and *Error* messages and allowed the user to specify the exit condition of the testbench.

Based on the above work the thesis is organized into following chapters

- Chapter 2 provides an overview of ASIC based implementation of the TOE and the need for a solid verification environment.
- Chapter 3 talks about the traditional verification environment and the advantages of using a C++ based environment in conjunction with CPeer.
- Chapter 4 describes the CPeer desgin based on object oriented design methodology and the CPeer related classes.
- Chapter 5 talks about various CPeer capabilities and explains the error injection capabilities of CPeer.
- Chapter 6 draws conclusions about the work done.

Chapter 2

The TOE Approach To High Performance Network Interface Design

Network interface design has received considerable attention in recent years. The growing interest can be attributed to the observed gross disparity between the raw bandwidth available from emerging gigabit networks and the maximum effective throughput that can be delivered to the end applications. It is a well known fact that memory bandwidth limitations coupled with overhead of operations such as data copying, software checksum computation, servicing of interrupts and context switching [15][16][17] are usually responsible for the observed poor performance. The Transport Offload Engine embraces several innovative features and integrates these pieces into a coherent whole to address these problems.

Figure 2.1 shows a high-level block diagram description of TOE enabled Network interface. As shown in the diagram the device is composed of three major functional blocks. These blocks are the full Transport Offload Engine, 10/100/1000 capable MAC, and a gigabit PHY. Support functions such as exception processor, system bus controller, Memory controllers, DMA Engine Controllers are also shown. Following sections describe each of the blocks.

2.1 The PHY

The PHY governs the hardware connection and the transmission of bit streams over a communication channel. The transmitting unit of this layer is a bit. It interfaces directly to twisted pair media via an external transformer.

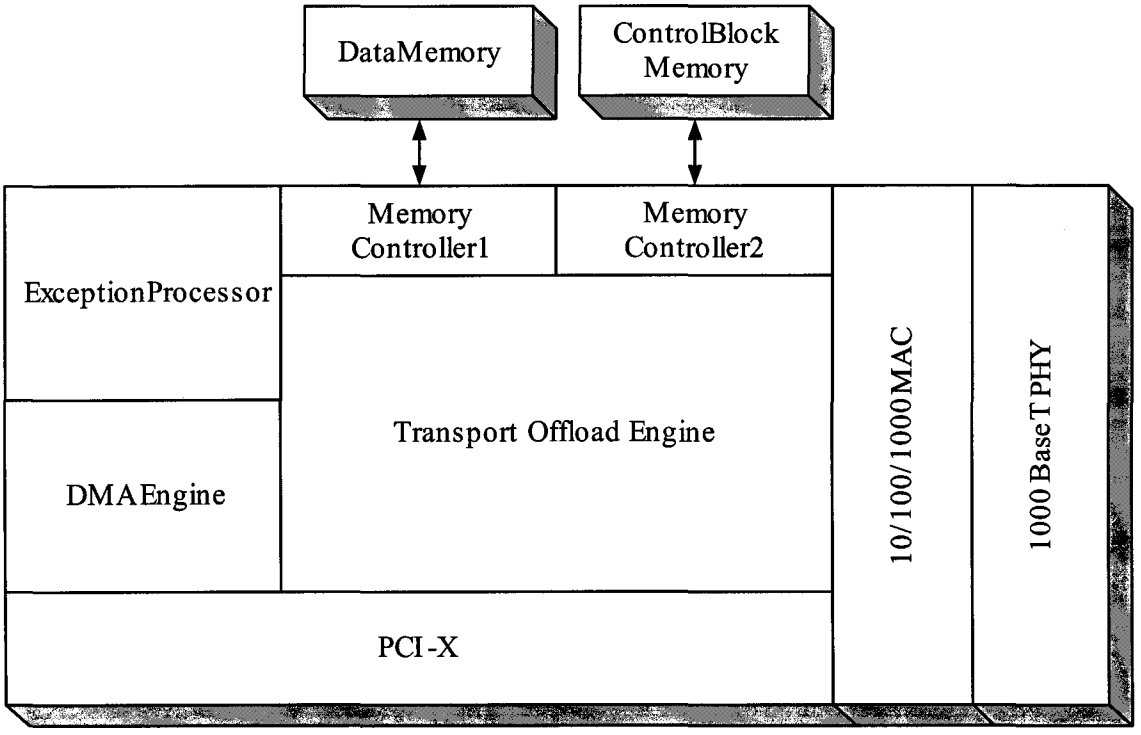


Figure 2.1 TOE Based Network Interface Controller

2.2 The 10/100/1000 Media Access Controller (MAC)

The transmitting unit in this layer is a frame. A frame is a stream of bits that flows across the Ethernet. This layer defines the rules for sending and receiving information from one node to another. It breaks the raw bit streams coming from Physical Layer into data frames, then transmits the frames sequentially and processes the acknowledgement

sent back by the receiver. It performs data packet exchange with the network through Media Independent Interface (MII/GMII). It provides a MAC address (hardware address) which can be used to identify each node in the network. Each MAC address has 48 bits. Each node in the network should have a unique MAC addresses, and MAC address is the only address that hardware devices can understand when frames are exchanged at the hardware level. It also provides the error detection, data link error recovery and modification.

2.3 The Transport Offload Engine

The transport offload engine supports the hardware acceleration of IPV4, TCP/UDP and ARP/RARP. Figure 2.2 shows the interaction between various modules of a TOE.

- The MAC interface module in TOE communicates with MAC, ARP and IP modules. The receive side of the module parses the Ethernet header, enables the next encapsulated protocol (ARP/RARP, IP etc) based on the type field in Ethernet packet header and aligns the packet data for the upper protocols. The transmit side processes the transmission request from the ARP/RARP, IP, TCP/UDP modules and forwards the packet to MAC based on the quality of service associated with the request. Care is taken such that none of the upper layer module is completely starved from transmitting a packet.

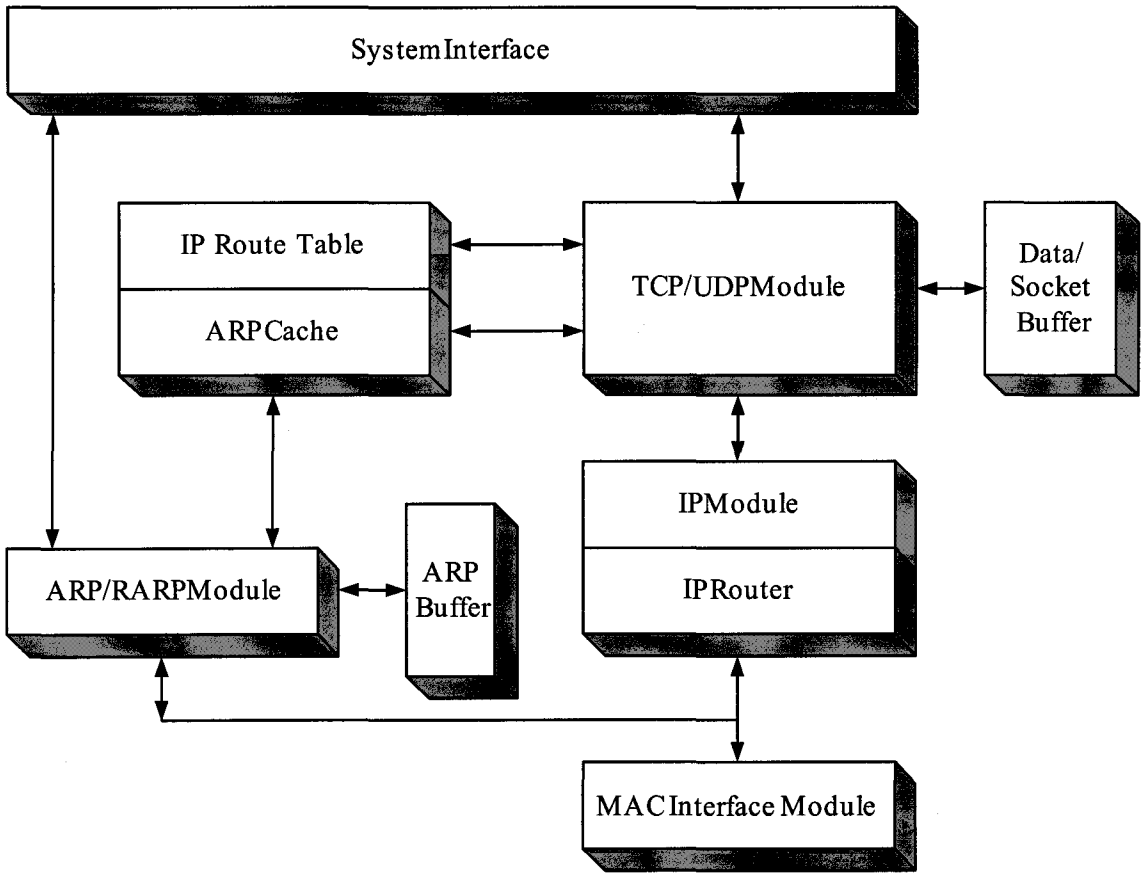


Figure 2.2 Block Diagram of a Transport Offload Engine

- The ARP module in the TOE responds to ARP requests by generating ARP replies. It can also generate the ARP requests in case of ARP cache misses. The RARP module generates the RARP request for an IP address.
- The IP module in the TOE detects if the received packet is of the supported type(IPV4 or IPV6) and silently drops a packet if it is not of the supported type. For supported type of packets the module parses the received packet for various fields and forwards this data to the higher layer protocol modules(TCP and UDP). It also generates the ICMP echo reply packets and handles the reciption of ICMP packets. It verifies the IP checksum of the received packets and generates

the IP checksums for outgoing packets..

- The IP Router module is used when host wants to transmit an IP packet. The host has to determine where to send the packet – either to the host on local area network, to an external network or back to the host itself. It is the task of the IPRouter to direct outgoing IP packtes to the appropriate host. When a transmitting module requests a route, it passes the destination IP address of a packet to the IP router. The router then compares the targeted IP address with a list of destinations stored in the IP route table. If a match is found, the IP router then attempts to resolve an appropriate Ethernet address. The router performs this resolution by requesting an ARP lookup in the ARP cache. If the destination Ethernet address is resolved it is passed back to the transmitting module which uses this Ethernet address as the destination for the outgoing Ethernet frame.
- The TCP module implements the TCP protocol. It is a connection-oriented protocol and provides a reliable service. It initializes a session before transferring any data to another node. It begins to transfer the data after assuring a handshake. The handshaking makes sure that the other endpoint is online and it is ready to accept the data. It closes the session when it is over. It is responsible for the error checking and correction. Each packet has a sequence number. Once the packet arrives at the destination node successfully, the destination node sends an acknowledgment back with the sequence number of this packet to the source node. If the source node does not receive the expected acknowledgement in time, the source node assumes that the packet is corrupted or get lost. The source node then retransmits the same packet. TCP can reconstruct the message with out-of-

order packets. If the packet arrives out-of-order, the destination node can reconstruct the whole message with the help of the sequence number of each packet. The TCP module supports delayed ACK, selective ACK, and out of order TCP packets. Protection against wrapped sequence numbers (PAWS) and the sliding window protocol is also supported

- UDP is connectionless protocol. UDP does not open a session before transmitting. It begins to transfer data without knowing the status of the destination node. UDP does not care if the packest reaches its destination or not. The packet can be lost or duplicated. The wrong packet is not corrected. The destination node can detect the packet error by checking the checksum of the receiving packet, but the destination node does not give any acknowledgement back to the source node. The source node never knows if a packet is lost, duplicated or corrupted. There is no recovery. Although each UDP packet contains a sequence number, the destination node doens't sort the out-of-order packets. UDP does not slow down its transmission speed when the network is congested. The speed at which UDP sends the data is only constrained by the rate at which the application generates the data, the capabilities of the source and the access bandwidth to the Internet. The impact is that more packets are lost when the network is congested and they can never be recovered.

2.4 The Exception Processor

An on-chip processor is provided for non-time-critical protocol processing and data steering. This processor is also equipped with all the peripherals needed to complete

a processor sub-system. Under normal operating conditions, the on-chip processor controls the TOE. It has the capability to address the memory located within data memory (Figure 2.1), the auxiliary processor peripherals and a local RAM. The RAM is used for instruction and data. The local memory is used for exception handling, and for the auxiliary processor to build raw packets that may be transmitted or received by the TOE.

2.5 The DMA Engine

The DMA Engine is made up of two independent TX and RX DMA engines. Together with the two DMA engines, there are two independent FIFOs for transferring data to/from the system interface and from/to the network. The FIFOs, providing temporary storage of data, free the host system from the real-time demands of the network. The way in which the FIFOs are emptied and filled is controlled by FIFO threshold values stored in internal configuration registers. These values determine how full or empty the FIFOs must be before the device requests the bus. Additionally, there is a threshold value that determines how full the transmit FIFO must be before beginning transmission. Once the TOE enabled network interface requests the bus, it will attempt to empty or fill the FIFOs as allowed by the settings in the TX and RX DMA configuration registers.

2.6 The Memory Controllers

Two banks of memory are used. One bank is used for packet buffer memory. It is used to store transmit and receive packets. While the other one is used to store the control information associated with the sockets.

2.7 The System Bus Interface

PCI stands for Peripheral Component Interconnect [10]. The PCI bus is a 32 or 64 bit system bus with multiplexed address and data lines, at operation speeds of 33, 66, 100 and 133 MHz. The bus is used as an interconnect mechanism between PCI controller component and processor/memory systems. PCI supports burst transfers. A burst transfer is the one consisting of a single address phase followed by two or more data phases. The start address and the transaction type are issued during the address phase. The target device latches the start address into an address counter and is responsible for incrementing the address from data phase to data phase.

Thus the TOE enabled network interface is a system-on-a-chip design with far more complex functionality and domain diversity. This brings with it challenges to traditional verification approaches. The next chapter describes the traditional verification approach and introduces CPeer, a C++ based verification environment that was developed to verify the TOE functionality.

Chapter 3

The CPeer Verification Environment

3.1 Verification Methodology

Figure 3.1 shows a conventional ASIC based functional simulation methodology. The methodology assumes the Register Transfer Level (RTL) description of a design as the input. RTL is a design description in which storage elements are interconnected by sections of combinational logic. The RTL code goes through the lint checking for syntax and synthesizability checks. The testbench is created based on the functional specification of the design. Then the functional verification is performed using event based [12] or cycle based [12] simulator.

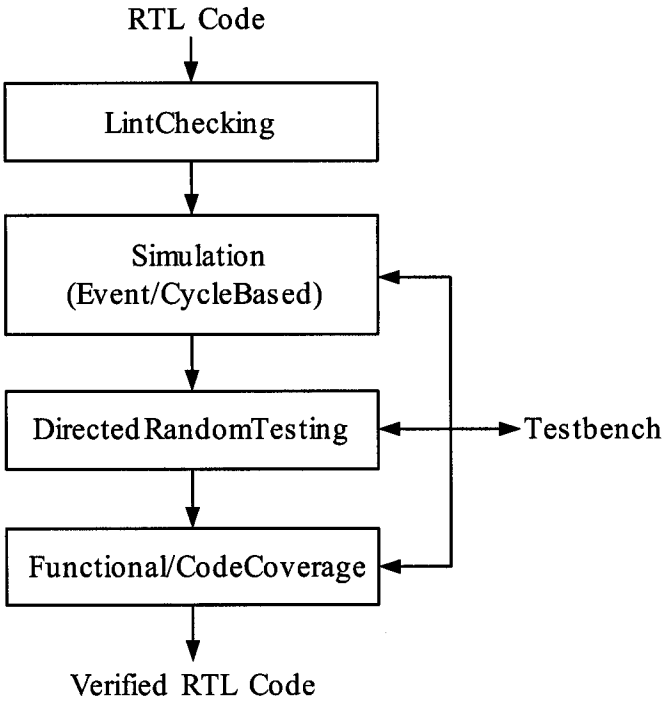


Figure 3.1 Functional Simulation Flow

A testbench is a layer of code that is created to apply input patterns to the design under test (DUT). The process of creating a testbench for a design involves thorough understanding of the functional specifications. The testbench that generates stimulus, samples the output of the DUT and compares the output with the expected (golden) results is called a self-checking testbench[12]. A self-checking testbench generates errors if the output does not match the expected results and provides an easy way to detect, understand and fix errors. This helps to analyze the DUT's functionality details and isolating the cause of error.

In the traditional verification techniques a testbench is created using a standard Hardware Description Language (HDL), such as Verilog [11] or VHDL [11]. HDLs have improved hardware design by allowing high-level hardware descriptions. However, because hardware verification needs some sophisticated mechanisms that HDL languages do not possess (e.g. complex data structures, object oriented design capabilities), HDL languages are not ideal for verification tasks. The HDL testbench becomes complex and difficult to maintain as the design verification requirements increase. Since a testbench does not have to be built in silicon, higher-level language like C++ [14] can be used for writing testbenches. C++ is a good vehicle for testbench capture. It is more suited for testcase generation. It is useful when we want to create complex test scenarios. Besides it gives a better flexibility when it comes to play tricks that make debugging easier and quicker. Following are some of the advantages of using C++ to verify the TOE design.

- Complex testcase generation with many activities going on simultaneously (e.g. a testcase with Exception processor, Host and PHY traffic all at once, rather than in sequence). The embedded processor code runs in multiple

threads, so it is like multiple testcases running simultaneously. In Verilog it is very tough to produce this kind of behavior. In C++ it is easier to achieve this with multiple threads running.

- Availability of abstract types like FIFOs, Stacks etc. to mimic the memories.
- Object Oriented test writing capabilities. It means greater versatility and re-usability. An example of this is simulating a testcase with multiple NICs. This can be done by encapsulating the test environment into a new object and then instantiating it as many times as needed.
- Better performance (speed) on calculation (e.g. checksums), easier modeling, expected value generation, expected value checking. For example, parsing the status message queue where each message has different length depending on its type. It is easier to implement a parser in C++ that checks for an expected message. But in Verilog the same task is rather hard to do.
- Better coverage. By writing lots of C++ monitors, all running quickly in C++, that automatically checks for expected behavior (not necessarily *exact* data values) one can move the testing from simple point feature testing to automated testing. By writing simple point feature tests as monitors, better coverage of point features is possible during exhaustive random testing. Take the Status Message Queue example from above. If this point test is written as a monitor, then it is not much work to make it monitor any random vectors thrown at it.

- Expandability. There's lot of public domain code that can be leveraged to generate different kinds of packets thereby simplifying the testcase generation.
- Quick testcase compile times. When the testcase is modified, only the C++ testcase code needs to be recompiled, not the Verilog RTL.

Thus as a base we can do everything we want in Verilog using C++ and much more.

3.2 The CPeer Model

CPeer is a C++ port of a public domain C code of TCP/IP network stack [6]. It emulates a 'peer' networking device, but is controllable and observable from the test program. The Peer model is shown in the Figure 3.2. It has a streamline architecture in which the packet flows from one layer to another and at each layer an appropriate header is added or stripped-off depending on the direction of packet flow. The Ethernet (with ARP), IP, ICMP, TCP and UDP layers of CPeer implement the corresponding protocol processing functionality. Internally CPeer has a TCP state machine [6] that arbitrates between different valid states depending on the type of received TCP packet. The model itself is event driven and supports the following capabilities:

- Initiate and close a TCP connection
- Produce responses (by way of the RFCs) to a number of different in-bound packet types (TCP, UDP, ICMP, ARP).

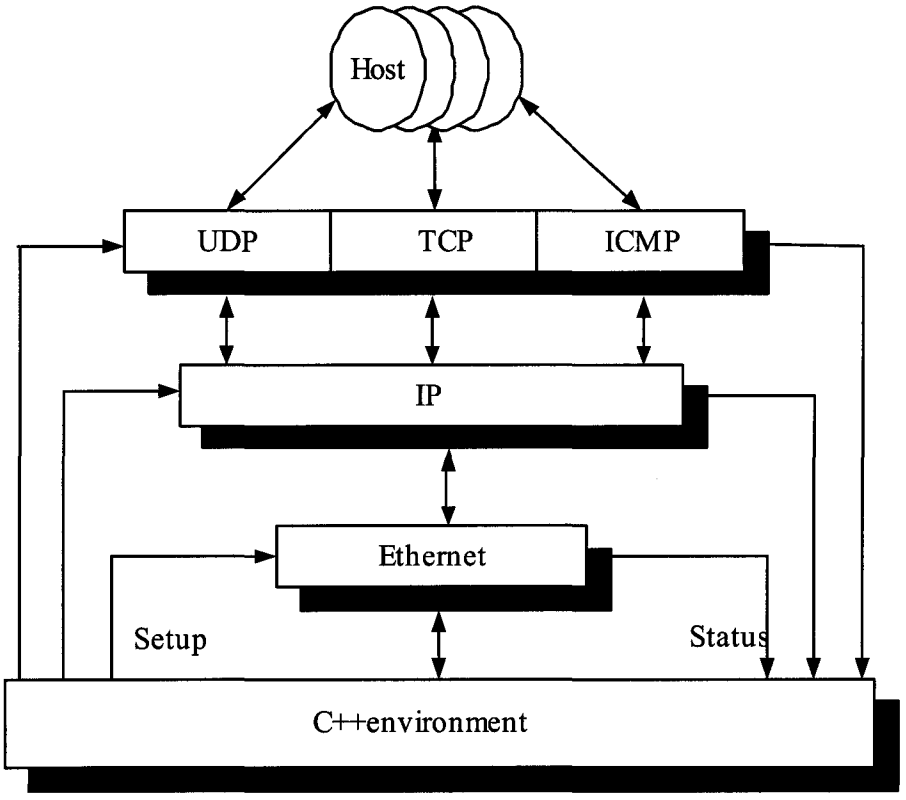


Figure 3.2 CPeer Model

- Transfer (random) data of varying size payloads to and from the stack.
- Maintain correct state information with multiple simultaneous TCP connections.
- Inject errors on packets sent to the DUT such as incorrect TCP checksums, IP checksums, UDP checksums, truncate packets at either end of an Ethernet frame, incorrect source and destination IP addresses, incorrect sequence numbers, incorrect ACK numbers, and modify the TCP flags within the TCP header. In essence the CPeer has the ability, under full user control, to modify any field within an IP header, TCP header, or UDP header.
- Monitor traffic in bound from the DUT for things such as a valid IP checksums, TCP checksums and framing errors.

- Visibility of all packet Header, Connection State and Data parameters to the simulation environment.
- Control and modification over the protocol (TCP/UDP/ICMP/ARP) behavior.

3.3 A Connection Queue in CPeer

A basic concept in the design of CPeer class is a *Connection{}* queue. Figure 3.3 shows a two-packet connection queue. A *Connection{}* queue is used for two purposes

(i) To store the control and status information of a connection between CPeer and TOE.

(ii) To store the packets those are received from the TOE and to store the packets those are transmitted to the TOE.

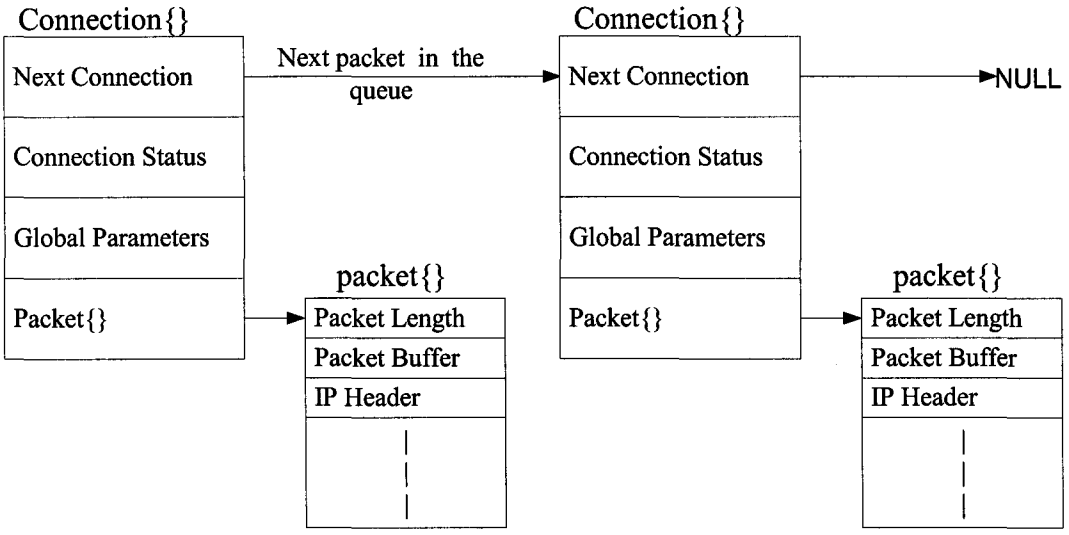


Figure 3.3 A Two Packet *Connection{}* Chain

For example the control and state information for a TCP connection is:

- Source and Destination IP addresses
- Source and Destination ports

- Sequence number
- Acknowledgment number
- Maximum Segment Size (MSS)
- Window size advertised.
- State of TCP state Machine

For a UDP connection the control information

- Source and Destination IP addresses
- Source and Destination ports
- Length of payload received successfully

For an ICMP connection the control information is

- Number of ping replies received successful
- Length of ICMP payload received.

The *packet{}* type data structure is the placeholder of packet header and payload. Each layer use this *packet{}* format to add the packet parameters/headers. The same packet format is used to transmit and receive packets. The detailed packet format is shown in the Figure 3.4.

The control and state information for each type of connection (TCP, UDP, ICMP) is visible from the test program. This gives the ability to control the protocol behavior from a test program and thus enables the Request for Comments (RFC) based testing of the TOE.

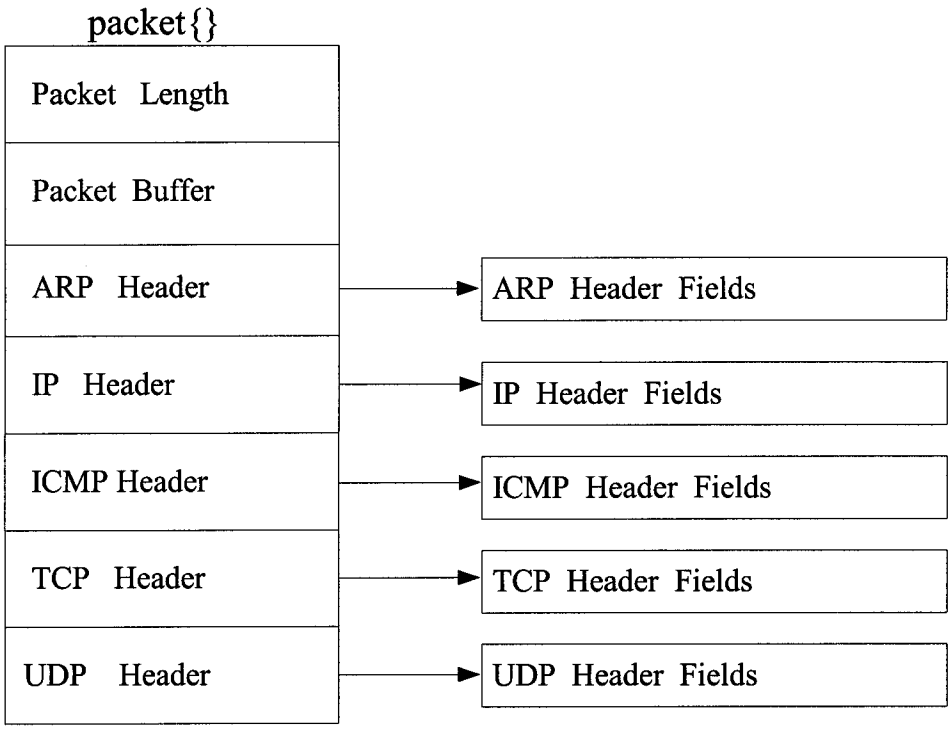


Figure 3.4 A *packet{}* structure

Each layer in CPeer has an independent receive and transmit queue. A layer continuously scans for a packet in its designated queue.

Chapter 4

The CPeer Structure

To accurately verify a design, ideally it should be placed in the real operational environment. But in general such a circumstance cannot be completely reproduced by simulation. Usually a testbench is written to:

- Emulate the environment of design under test (DUT) (In our case TOE)
- Self check the response of TOE to the stimuli generated by the environment.
- Be easily configurable to allow test writing by users unfamiliar with verification environment.

One of the major objectives of the CPeer is to provide a simple and flexible environment to verify the functionality of the TOE. Figure 4.1 represents the functional view of the simulation environment. It shows the main functional verification components that are used to stimulate the TOE and to verify its behavior. The TOE RTL (DUT) also called as Network Stack RTL sits at the center. CPeer acts as both a target and an initiator agent for verifying the TOE functionality. The *Virtual MAC Model* acts as a MAC with a unique MAC address. The *MAC Model* drives and snoops a Gigabit media independent interface (GMII) for the packets. The TOE and CPeer model are configured to the desired state before initiating the test.

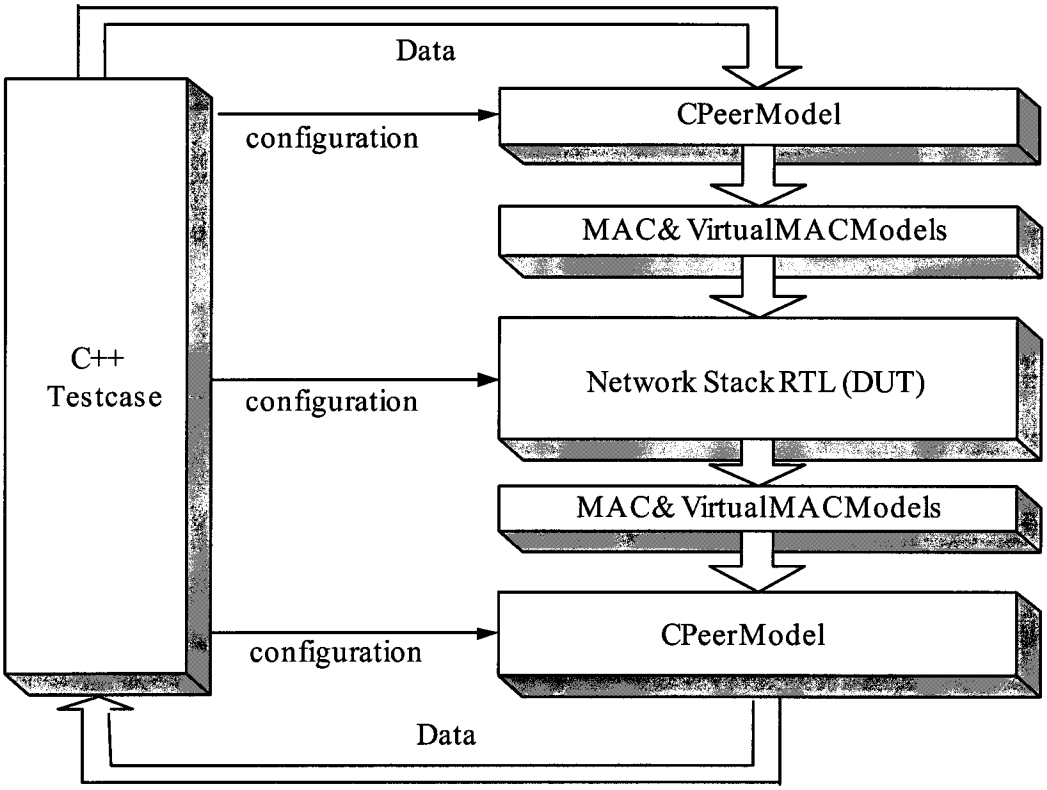


Figure 4.1 CPeer Based Functional Environment

4.1 Overview of CPeer Related Classes

The CPeer functional simulation environment is based on the object oriented design principles. It is a base for the development of test programs to verify the functionality of TOE. The simulation environment is divided into three parts i.e. the TOE RTL to be verified, the interface module that interacts with TOE and the CPeer model that is used to generate the stimulus. Each functional verification component outlined in Figure 4.1 is associated with a class within the code structure. At the highest level of the environment there are three classes involved in setting up CPeer.

- *MAC* class
- *VMAC* class

- *CPeer* class

The *MAC* and the *VMAC* class are the interface classes that interact with TOE on one side and CPeer on the other side. They snoop the packet traffic on the Ethernet link and filters them based on the MAC address. The *CPeer* class is responsible for stimulus generation. It handles packet processing and generation of desired response by invoking the appropriate protocol (TCP,UDP, ICMP, ARP).

In the following sections, each class is carefully outlined and its relationship to the other classes is described in more detail.

4.1.1 The *MAC* class

The *MAC* class connects to a Verilog bus functional model (BFM) of a MAC/PHY. A bus functional model is the behavioral representation of a unit, which interfaces with a DUT (TOE). It raises the level of abstraction and does protocol checking at the DUT interface. The purpose of *MAC* class is to create a link between the *VMAC* class and the TOE interface. In other words this class creates a connection with the HDL code of the TOE. It has no MAC address, but can output packets and snoop all incoming traffic. Therefore a MAC class instance can function as a collection of NICs (Network Interface Card)

The *MAC* class has methods to control the mode in which MAC can operate viz 10,100 or 1000 Mega Bits per second. It is also possible to control the Inter-Frame Gap (IFG) while transmitting or receiving the frames. One can set the timeout on the MAC trying to send a packet.

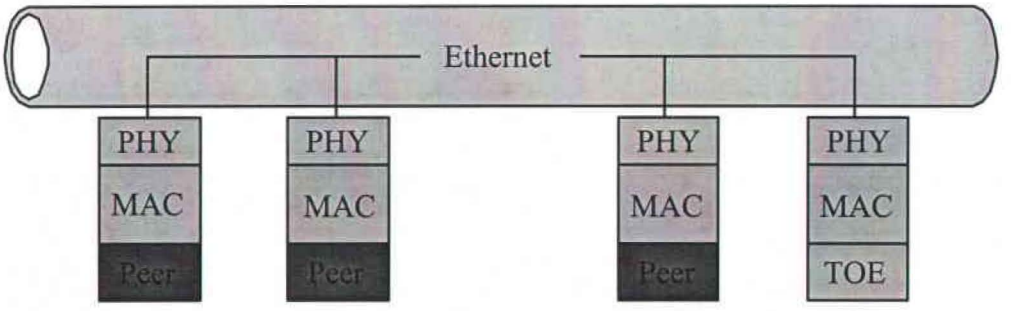
4.1.2 The *VMAC* class

The *VMAC class* is also known as the *Virtual MAC class*. It connects to a *MAC class* instance so that it can send and receive packets. Unlike the *MAC class*, however, it does have a unique MAC address. So it acts as a single MAC, which rests upon an instance of the MAC class. Many *Virtual MACs* can be registered with a single *MAC class* instance. When registered, a *Virtual MAC* receives a filtered portion of traffic from the *MAC*, and may inject new packets to the *MAC*. The *Virtual MAC* may change the filtering as needed, allowing for multicast snooping, or may even snoop all traffic. For each packet the MAC receives from the Verilog BFM, it examines the Ethernet address and queries each registered VMAC to see if it is interested. If it is, then it passes the packet up to the interested VMAC instance.

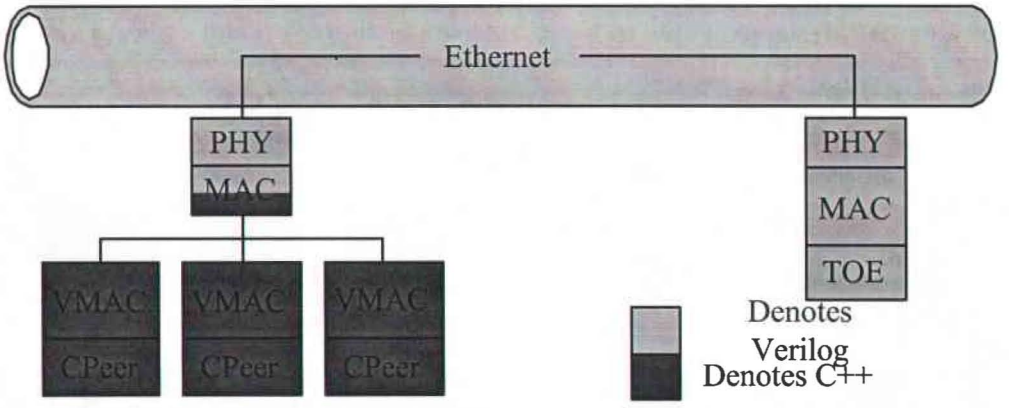
The *VMAC* class has the methods to add the MAC address to the list of address being snooped. It is possible to program a *VMAC* instance to snoop all the incoming frames. One can enable and disable the link using the *VMAC* methods. This enables the testing of drop packet cases.

Figure 4.2 gives a more visual idea about these classes. It illustrates a simulation environment in which multiple Peers are communicating with the TOE on an Ethernet link. At the top of the diagram we see a Verilog simulation environment with multiple MAC BFMs instantiated. Each Verilog BFM has a unique MAC address and each one is injecting and snooping data to the TOE. So if we have to increase the number of Peers in the simulation environment then we have to change the Verilog BFM of MAC and instantiate it again. This leads to recompiling the Verilog code, which is time consuming. At the bottom of the diagram we see a simulation environment, which is functionally

equivalent to the top one. In it a single MAC BFM with not MAC address is connected to multiple peers (*VMACs*) having a unique MAC address. Here it is possible to dynamically grow as much we much as we are allowed and as much as memory we have and as much Peers as we want This is important because it means that the number of Peers running in the simulation environment can vary without modifying the HDL to explicitly instantiate a MAC BFM for each Peer.



Three MAC-PHY BFMs talking with the TOE



One MAC-PHY BFM talking with the TOE with three 'virtual' MACs

Figure 4.2 A Multi-Peer Simulation Environment

4.1.3 The *CPeer* class

The *CPeer* class is a core class that handles all the packet processing. The instance of this class connects to a *VMAC* instance. *CPeer* uses the *VMAC* instance to

send and receive packets with particular MAC addresses. It also supports multiple instantiations, which reduces the effort in creating a multiple Peer simulation environment. In CPeer the processing of TCP, UDP and ICMP protocols is handled by the respective *Connection* classes. All the *Connection* classes are the *friends* of *CPeer* class and they can access its private and protected members. This gives the ability to control the individual protocol behavior in CPeer from the test program.

4.2 Working with Connections

After setting up CPeer, all that is needed is to create and manipulate connections on both the CPeer side and the TOE side. CPeer primarily supports three types of connection viz. ICMP, TCP and UDP. The *Connection* class is the base class for all the connections viz ICMP, UDP and TCP. The *Connection* class has methods that can access and manipulate the IP header fields. All ICMP, TCP and UDP Connections share these methods when it comes to manipulating the IP header fields. It provides common methods for sending and receiving packets using a *CPeer*.

Following sub-sections describe each type of *Connection* class and the associated methods

4.2.1 *ICMPConnection* Class

As the name suggests this class is used for verifying the ICMP ping response of the TOE. It has the methods to generate the ICMP echo requests (ping) and verify the correctness of the ICMP echo replies received from TOE. It has methods to set and return the values of various ICMP header fields [6]:

- ICMP Packet Type: An 8 bit integer message type

- ICMP Code: An 8 bit code that provides further information about the message type.
- ICMP Checksum: A 16 bit checksum field that covers the ICMP message
- ICMP Identifier and Sequence Number: These fields are used by the CPeer to match the replies to requests.

The way an ICMP Connection is setup in the testcase is as follows:

First an instance of ICMPConnection type class is created by registering it with the CPeer instance.

```
ICMPConnection myconnection(CPeer, peerIPadd);
```

This allocates the resources within CPeer that are required for maintaining the ICMP conncetion. Then the connection is manipulated using the various methods of the class. The methods are described in detail in Appendix A. For example following code sets up various IP header fields and sends a ping packet to TOE.

```
char buf[0x10] = {0x3c,0x72,0xce,0x3b,0x9b,0xb1,0x0b};
myconnection.setFragmentOffset (0x4000); // set IPRAG bit
myconnection.setTimeToLive (0x40);
myconnection.ping(0x0738, 0x0000, 0x38, buf); // send PING
packet
```

4.2.2 *TCPConnection* Class

This class is used for verifying TOE's TCP functionality. The connections are somewhat more complicated in this class. It has the methods to setup the client or server sockets and verify the correctness of the packets received on each socket. For each socket it is possible to control the TCP protocol behavior by manipulating the various TCP

options in the header field. This class has also the methods to set and return the values of various TCP header fields [6]:

- 16-Bit Source and Destination ports
- 32-Bit Sequence Number
- 32-Bit Acknowledgment Number
- 4-Bit TCP Header Length
- 1-Bit TCP Flags: URG, ACK, PSH, ST, SYN and FIN
- 16-Bit Window Size
- 16-Bit TCP Checksum
- 16-Bit Urgent Pointer and
- TCP Header Options of variable size (Maximum Segment Size, Window Scale, Timestamp, Selective Acknowledgment, No Operation, End of Operation list)

The way a TCP Connection is setup in the testcase is as follows:

On the CPeer side the following code *instantiates* a *TCPConnection*:

```
TCPConnection  tcpsocket(CPeer, peerPort, NSIPadd, NSPort);
```

This is different from *establishing* a TCP Connection. Instantiating a connection does not send any traffic but it does place the connection in the LISTEN state on the CPeer side. The following code sets up a client socket with the TOE sending a SYN packet:

```
tcp.open_client_socket(NSIPadd,peerIPadd,NSPort,peerPort,
rx_start_add_val,rx_buffer_len_val,socket_handle_value);
```

The CPeer then automatically respond with an ACK. It then waits for a SYN+ACK and enters the ESTABLISHED state. We can wait for this event to occur with:

```
tcpsocket.waitTcpState(TCPS_ESTABLISHED);
```

To make sure that the TOE is also in the ESTABLISHED state we try:

```
tcp.verif_socket_est(100, active_socket_handle);
```

This waits for a connection to become established and returns the socket handle. We check this against the real socket handle to make sure it's really our socket that is currently active.

```
if ( !Eq(active_socket_handle, socket_handle_value) ) {
     display("ERROR: Active socket handle: 0x%h  Expected:
0x%h\n",active_socket_handle, socket_handle_value);
     finish();
  }
```

Alternatively we can have CPeer initiate the connection. To do this we first prepare a socket on the Network Stack (NS) side:

```
tcp.open_server_socket(NSIPadd,NSPort,rx_start_add_val,rx_b
uffer_len_val);
```

Then we have CPeer send the initiating SYN packet:

```
tcpsocket.sendSyn();
```

The Network Stack should automatically respond with an SYN+ACK packet. CPeer will then automatically respond with a ACK packet. And the connection would thus be established. Again, we can check that both sides are in the established state by using the methods *waitTcpState()* and *verify_socket_est()* as detailed above.

Once a connection has been established we can manipulate the CPeer to send and receive data. It is very simple to send and receive data on the CPeer side:

```
// Generate 4096 bytes of random data
DynamicBufT randomdata = genRandomData(0x1000);
// Queue it up for auto-send
```

```
tcpsocket.sendData(randomdata);
DynamicBufT receivedData;
tcpsocket.receiveData(receivedData);
```

However, we must also maintain the TOE side of the connection.

The functions *NSsendData2peer* and *peerSendData2NS* setup both the TOE and Peer side, transfer the data, and then compare the received contents with the intended values. For example:

```
NSsendData2peer(tcpsocket,active_socket_handle,randomdata);
```

This will transfer the contents of *randomdata* to a predefined area in Host memory, then have the Network Stack DMA chunks across and transmit packets. As the data is received on the CPeer side, the contents are checked against the original data, *randomdata*.

Likewise:

```
peerSendData2NS(tcpsocket,active_socket_handle,randomdat,
window_send_size,urgent_pointer,rx_start_add_val,rx_buffer_
len_val, userflags);
```

This will transfer the contents of *randomdata* to the Host memory buffer at location '*rx_start_add_val*' with size '*rx_buffer_len_val*'. As the data is received on the Network Stack side, the contents are similarly checked against the original data, *randomdata*.

4.2.3 *UDPConnection* Class

This class is used for verifying TOE's UDP packet processing. It has the methods to setup the UDP connection and verify the correctness of the packets received on each

connection. It has also the methods to set and return the values of various UDP header fields [6]:

- 16-Bit UDP Source Port
- 16-Bit UDP Destination Port
- 16-Bit UDP Message Length and
- 16-Bit UDP Checksum

Creating and manipulating an UDP Connection is simpler than a TCP Connection. It is done in following way in a testcase:

On the CPeer side the following code *instantiates* an *UDPConnection*:

```
UDPConnection udpsocket(cpeer, peerPort, NSIPadd, NSPort);
```

This initializes a UDP Connection queue in the CPeer. The function *peerSendData2NS* can be used to transfer the data, and then compare the received contents with the intended values.

```
peerSendData2NS(udpsocket, SocketHandle, sendbuf,
rx_start_add_val, rx_buffer_len_val);
```

Chapter 5
CPeer Capabilities

This Chapter describes in detail the various capabilities of CPeer that are useful in verifying the TOE functionality and creating different test scenarios.

5.1 CPeer as a Packet Generator

CPeer has the capability to generate ARP, ICMP, and TCP/UDP packets. It enables custom packets with configurable Ethernet, IP, TCP, and UDP layers as well as custom payloads. One can command CPeer to transmit a specified type of packet and it can also respond automatically to a received packet. For example if one wants to test the TOE functionality to create a TCP connection then one can command CPeer to transmit a SYN packet. The TOE should respond with SYN-ACK packet. CPeer will then automatically send a ACK packet, thus completing a three way handshake and establishing a TCP connection.

Following tables shows the capability of CPeer in terms of header generation while transmitting a packet. The header fields in a packet take the values depending on the state of the connection. However it is possible to override any header field value by means of special function calls from a test case.

Table 5.1 IP Header Generation

IP Header Field	Generation
IP Version & Header Length	Only supports IPv4 headers with a length of 20 bytes (id 0x45)

IP Header Field	Generation
Type Of Service	Default is 0x00, but it is programmable from the test case.
Total Length	Fills up the appropriate packet length
Identification	Default is 0x00, but it is programmable from the test case
Fragmentation Flag/Offset	Default is 0x00, but it is programmable from the test case
Time To Live	Default is 0xFF, but it is programmable from the test case
Protocol	Supports: ICMP, TCP & UDP. IGMP is not supported.
IP Checksum	It automatically computes the IP Checksum.
IP Options	By Default no IP options are set, but they can be added to an outgoing packet.

Table 5.2 TCP Header Generation

TCP Header Field	Generation
Source/Destination Ports	Programmable from the test case.
Sequence Number	Programmable from the test case.
Acknowledgement Number	Programmable from the test case.
Header Length	Supports header length of fixed size (20 bytes).
Flags	Supports ACK, PSH, RST, SYN and FIN flags. URG is not supported. Programmable by the test case.
Window Size	Default is 0xFF, but it is programmable by the test case
TCP Checksum	It automatically computes the TCP Checksum.
Urgent Pointer	Programmable by the test case.

TCP Header Field	Generation
TCP Options	Programmable by the test case.

Table 5.3 UDP Header Generation

UDP Header Field	Generation
Source/Destination Ports	Programmable while initiating a UDP connection in the testcase.
Length	Automatically computes the appropriate header length.
UDP Checksum	It automatically computes the UDP Checksum.

Table 5.4 ICMP Header Generation

UDP Header Field	Generation
Type	Fully supports the Request & Reply type packets. It is also possible to program the type field to any desired value.
Code	Default is 0x00, but it is programmable by the testcase.
ICMP Checksum	It automatically computes the ICMP Checksum.

5.2 CPeer as a Packet Analyzer

Following tables show the capability of CPeer in terms of header processing of a received packet. CPeer takes appropriate actions based on the values set in the packet header fields. This gives us an ability to test TOE's response to abnormal connection cases. CPeer has the capability to analyzes the following header fields for an incoming packet:

Table 5.5 IP Header Analysis

IP Header Field	Analysis
IP Version & Header Length	Only supports Ipv4 headers with a length of 20 bytes (id 0x45)
Type Of Service	Ignored.
Total Length	Computes the total packet length.
Identification	Ignored.
Fragmentation Flag/Offset	Ignored.
Time To Live	Ignored. If the received packet has a TTL of 0x00 then it is NOT discarded. Also no ICMP error message is generated.
Protocol	Supports: ICMP, TCP & UDP. IGMP is not supported.
IP Checksum	Reports an error in case of incorrect Checksum.
IP Options	Takes actions based on the type of option received.

Table 5.6 TCP Header Analysis

TCP Header Field	Analysis
Source/Destination Ports	It performs a check for the ports. If the ports are not found then CPeer generates an error.
Sequence Number	Checks for a valid Sequence Number.
Acknowledgement Number	Checks for a valid Acknowledgement Number.
Header Length	Supports header length of any size. If the TCP Options are set then they are ignored.
Flags	Supports ACK, PSH, RST, SYN and FIN flags..
Window Size	Takes care of the advertised window size.
TCP Checksum	Reports an error in case of incorrect Checksum.

TCP Header Field	Analysis
Urgent Pointer	Ignored
TCP Options	Takes action based on the type of option received

Table 5.7 UDP Header Analysis

UDP Header Field	Analysis
Source/Destination Ports	Checks whether or not the ports are valid.
Length	Computes the appropriate header length.
UDP Checksum	Reports an error in case of incorrect Checksum.

Table 5.8 ICMP Header Analysis

UDP Header Field	Analysis
Type	Supports the Request & Reply type packets. All other types are ignored.
Code	All codes other than the default (0x00) are ignored.
ICMP Checksum	Reports an error in case of incorrect Checksum.

5.3 ARP Processing

CPeer automatically responds to ARP requests packet with an ARP reply. If the desired entry is not found in its ARP table then a random one is generated. CPeer can also *send* ARP requests, but only at the bequest of a testcase. CPeer does not automatically try to resolve IP addresses into MAC addresses.

CPeer *ignores* all incoming ARP replies. Entries in its internal ARP table must be added manually. Entries in its internal ARP table must be written to and read from explicitly, and may very well be random if it has processed an ARP Request for that IP address already.

5.4 Error Injection Capabilities of CPeer.

The goal of CPeer is not to verify just TOEs response to legal packets but also to verify its response to illegal packets. CPeer has a unique capability to inject errors in an outgoing packet. So it is possible to corrupt any field in the packet header. This aids in verifying the TOE's response to non-standard packet traffic.

Corruption events are inserted by using the *corrupt** methods of Connection classes. The generic format of a corruption event call is:

```
corruptXXXX(Corrupt_Type type=REGULAR,
Corrupt_Random is_random = RANDOM, unsigned long value=0);
```

Defaults are specified for all three arguments, so it is possible to do a default random corruption by invoking something that looks like:

```
myconnection.corruptXXXX();
```

Alternatively each field can be explicitly defined:

```
myconnection.corruptXXXX(SMART);

myconnection.corruptXXXX(REGULAR,SPECIFIED, 0xdeadbeef);
```

Here are the possible values for each field:

First argument (Corrupt_Type): It can take three different values viz, REGULAR, PARTIAL and SMART.

- 'REGULAR' is the default corruption method. It corrupts the given field and recalculates the header checksums.
- 'PARTIAL' corruption allows the user to force a particular value or random value into the specified field without re-calculating the header checksums

- 'SMART' corruption refers to the case of recalculating the appropriate checksums so that the new field can be inserted into the outbound packet without causing a checksum violation.

Second argument (Corrupt_random): It can take two different values viz RANDOM and SPECIFIED.

- RANDOM is the default corruption method. It fills the field to corrupt with random data
- SPECIFIED. When this field is specified then the third argument values is used as a corruption field value.

Third argument:

This is only used if the second argument is set to SPECIFIED. See above.

Each *corruptionXXXX()* call adds that particular corruption to a list of corruptions to inject into the connection. In order to clear these corruption events the *corruptClear()* method is used.

e.g.

```
myconnection.corruptClear();
```

5.5 Setting Up CPeer Simulation Environment

The simulation environment is setup in the following way:

1. Instantiate a MAC
2. Instantiate a VMAC that uses the MAC, optionally give it a MAC address otherwise a random one will be chosen
3. Instantiate a CPeer that uses the VMAC.
4. Initialize the MAC BFM

5. Initialize the CPeer
6. Enable the link so that packets can be sent/received
7. Set CPeer's IP address
8. Set the parameters for the Design Under Test, namely the TOE's MAC address, and the TOE's IP address

Here is typical source code to do this:

```
Eaddr peerMACadd = 0xcafedeadbeefULL;
Eaddr NSMACadd   = 0x002233445566ULL;
IPaddr peerIPadd = 0x0a0a0a02;
IPaddr NSIPadd   = 0x0a0a0a01;
MAC         mac(hdlPath);
VMAC        vmac(mac, peerMACadd);
CPeer       CPeer(vmac);
mac.init();
CPeer.init();
mac.EnableLink();
CPeer.setIPAddress(peerIPadd);
CPeer.setDUTAddress(NSMACadd, NSIPadd);
```

Notice the types:

Eaddr this is an 'Ethernet Address' type

IPaddr this is an 'IP Address' type

The *Eaddr* type can have each byte be accessed by the 'pos' member:

So the above assignment to the *NSMACadd* variable is equivalent to:

```
NSMACadd.pos[0] = 0x00;
```

```
NSMACadd.pos[1] = 0x22;
NSMACadd.pos[2] = 0x33;
NSMACadd.pos[3] = 0x44;
NSMACadd.pos[4] = 0x55;
NSMACadd.pos[5] = 0x66;
```

The *IPaddr* type functions just like an *u32* (32 bit unsigned integer) type.

5.6 Tasks for Controlling the TOE

In order to verify the functionality of TOE, a testcase should also have an ability to configure various modules of TOE (TCP, UDP, IP, ARP, Ethernet interface, internal memories) to a desired state. To do this various tasks were developed that communicate with each module of the TOE from the Host interface. Following is a list of such tasks that were required for controlling the behavior of the TOE. These methods were common to most of the testcases.

- Create a TOE client socket to connect to a server without setting host buffer lists
- Create a TOE client socket
- Create a TOE server socket without setting the host buffer lists
- Create a TOE server socket
- Confirm that TOE socket register values are set properly
- Create a TOE UDP socket to connect to a server (with data passing)
- Create a TOE UDP socket to connect to a server (without data passing)
- Check the TCP_STATE of TOE socket for a given socket handle
- Access to multiple TOE
- Clear the TOE socket specific registers

- Request data from host memory be transmitted
- Check if transmit DMA has occurred successfully
- Check if the received packet is DMA'ed into host memory
- Set up buffers in host memory after receiving a connection to a server socket
- Initiate an ICMP request packet
- Add entries to the TOE ARP cache table.
- Add/remove entries from the route table, set maximum lifetime of ARP cache entry, generate ARP requests etc.
- Process the status messages generated by the TOE.

For explanation of each of these tasks please refer to Appendix B.

5.7 Debugging

CPeer can print out verbose information of what it is processing. This can assist in pinpointing a Stack problem. To enable this mode a plusargs argument should be used

+cpeer_debug

There is also a 'lower' level of printing below 'verbose' mode, which just prints packet analysis information and does not print CPeer's internal state information. This is enabled by: +cpeer_print

The +cpeer_debug option overrides the +cpeer_print option. In other words, there is no need to specify both these options.

CPeer can print out a lot of information detailing what is happening.

Here is an example C-Peer statement:

CM(eop): SENDING :	- Network Stack is sending, CPeer is receiving
%%==========ETHER==STAR T==========	- This is the incoming packet
CA FE DE AD BE EF 00 22 33 44	
55 66 08 00 45 00 00 2C 79 55	
00 00 FF 06 1A 60 0A 0A 0A 01	
0A 0A 0A 02 12 34 12 35 3D DD	
98 31 00 00 00 00 60 02 02 18	
73 80 00 00 02 04 05 B4 00 00	
81 4E 3A CE	
%%==========ETHER==STOP ==========	
CM:GEN2ETHERNET	- Transferring from 'Physical Layer' to 'Link Layer'
CM:Packet_type 800	- This is an IP packet
CM:Invoking STACK Send	- It is sending this UP the stack
CM:ETHERNET2IP2TRANSPORT	- It is going from the 'Link Layer' to the IP and then the Transport Layer
CM:ROUTING TCP PACKET	- It is at the TCP layer
CM:TCP PORT FOUND : 25	- It has found a matching connection for this packet with handle '25'
CM:LEN : 24 OFF : 24	- After stripping off lower layers packet is now

	24 bytes, with payload at offset 24
CM:LEN : 0	- The payload is therefore 0 bytes
CM:CPeer COMPUTED TCP CHKSUM : 0	- Making sure that subtracting the checksum in the packet with the real checksum is 0
CM:CPeer COMPUTED TCP CHKSUM : ffff8c43	- What the checksum should be
CM:IP source addr is: a0a0a02	- This is an important line to note (when +CPeer_debug is enabled). This marks the beginning of a response packet, rather than continuing analysis of a received packet
CM:IP destination addr is: a0a0a01	
CM:TCP source port is: 1235	
CM:TCP destination port is: 1234	
CM:TCP sequence number is: fffff0ff	
CM:TCP ack number is: 3ddd9832	
CM:TCP offset is: 50	
CM:TCP code is: 12	These are the TCP Flags. Here SYN and ACK are set.
CM:TCP window is: 1000	
CM:TCP checksum is: 8c43	
CM:TCP urgent ptr is: 0	
CM:TCP payload is: 0	
CM:TCP paylength is: 0	

CM:TCP2IP2PPP	- It is now sending this response down the stack from the TCP layer to the IP layer.
CM:ROUTING IP PACKET	- It is adding IP headers and sending to the 'Link Layer'
CM:SOURCE PORT FOUND : 25	- The IP packet is from handle number '25'
CM:ETHER2GEN	- It is now sending it out onto the 'Physical Layer'
CM:Packet Detected at Queue	- An packet is being seen coming out of CPeer
CM:Incoming Packet :	- It says "incoming" packet as it is "incoming" with respect to the Network Stack
%%==========ETHER==STAR T==========	
00 22 33 44 55 66 CA FE DE AD	
BE EF 08 00 45 00 00 28 00 00	
00 00 FF 06 93 B9 0A 0A 0A 02	- This is the packet being sent to
0A 0A 0A 01 12 35 12 34 FF FF	the network stack
F0 FF 3D DD 98 32 50 12 10 00	
8C 43 00 00 00 00 00 00 00 00	
==========STOP========	

What we see first is actually C-Peer receiving a packet, even though its commentary is "SENDING: " this is with respect to the TOE. We then see that it is in the section:

CM:GEN2ETHERNET

This is describing the transition of data from the 'physical' layer to the link layer. This is followed by a description of the packet type (whether it's a LOOP, ECHO, IP, ARP, RARP or Pause Frame – in this case an IP packet). It then states that it is "Invoking STACK Send" which is sending the packet *up* the stack. So on the next lines one can see:

CM:ETHERNET2IP2TRANPORT

CM:ROUTING TCP PACKET

Thus it has made it all the way up to the TCP layer. Having arrived, we are told:

CM:TCP PORT FOUND : 25

Here the number '25' is not the port number but rather a handle/identifier for a particular connection. To display this for a particular Connection, try:

```
display("myconnection's identifier is %s\n",myconnection.name() );
```

Note that at 'CM:IP source addr is: a0a0a02' the CPeer is responding to the incoming packet, in this case it is responding to a SYN packet with a SYN+ACK packet.

Chapter 6
Conclusions

CPeer was found very useful in pre-silicon verification of the TOE. It made the RFC based testing of TOE possible. It also helped in writing complex testbenches which otherwise is difficult to write using HDLs. With CPeer it was easier to separate the response checking and stimulus generation so that they can be done in parallel while running a testcase.

Due to its layered architecture and objected oriented base it is easier to expand the capabilities of CPeer. New set of protocol processing functionalities like iSCSI , IPsec, IGMP etc. can be added to CPeer.

One of the CPeer's key limitations is the lack of a true single step mode of operation. CPeer was written to automatically respond to a packet presented to it by the DUT. This behavior becomes problematic when one attempts to set up a particular condition within the DUT. To avoid it control functions can be added at each layer of CPeer to disable the auto response.

Appendix A

CPeer Methods

This Appendix explains in detail the methods available with each of the following base classes: *MAC*, *VMAC*, *CPeer*, and the derived classes: *ICMPConnection*, *TCPConnection* and *UDPConnection*.

MAC Class Methods

The MAC class connects to a verilog BFM of a MAC/PHY. It is derived from the verilog BFM

Constructor:

The constructor is called with the HDL-path to BFM MAC.

```
MAC(char *_hdlPath);
```

The MAC class methods are as follows.

Table A.1 MAC Class Methods

Public Method	**Description**
void init(int _speed=2, int _mii_type=2, int _duplex_status=1, int _IFG=96);	This task initializes the MAC, setting up a thread for handling incoming packets as well as initializing the verilog BFM. Default parameters are provided for a simple init() invocation. speed: 0 = 10 Mb/s, 1 = 100 Mb/s, 2 = 1000 Mb/s * default mii_type: 0 = reduced MII, 1 = full MII 2 = Gigabit MII * default 3 = Serial MII duplex_status: 0 = half-duplex 1 = full * default IFG: Inter-Frame-Gap Default of 96
bool AddVMAC(VMAC *vmac);	Registers a Virtual MAC. Returns 'false' if an entry already exists.
bool RemoveVMAC(VMAC *vmac);	Unregisters a Virtual MAC, returns 'false' if the entry can not be found.

Public Method	Description
bool isLinkActive();	Returns whether or not the Ethernet Link is 'up'.
void EnableLink();	Enables the Ethernet Link.
void DisableLink();	Disables the Ethernet Link. Note: if a packet is currently being transmitted at the Ethernet Link layer then that transmission will complete before halting further transmissions. Likewise for any currently receiving packet.
void sendPacket(DynamicBufT& buf, int IFG, int MACtimeout);	Sends a packet with a specific inter-frame-gap and timeout.

VMAC Class Methods

The VMAC class acts as a "Virtual MAC" with a *single* MAC address. It is capable of sending packets and snooping for packets with particular MAC addresses (or even snoop all packets). The VMAC class needs a MAC instance in order to provide these services.

Constructors:

Constructor with a MAC instance and a MAC address:

```
VMAC(MAC& mac, const Eaddr& MAC_address);
```

Constructor with just a MAC instance, and a random MAC address:

```
VMAC(MAC& mac);
```

Table A.2 VMAC Class Methods

Public Method	Description
void addMACSnoop(const Eaddr& MAC_address);	Adds a MAC address to the list of addresses being snooped.
void removeMACSnoop(const Eaddr& MAC_address);	Removes a MAC address from the list of addresses being snooped.
void clearMACSnoop();	Remove all snoops (but keeps looking for packets to its own MAC address and broadcast packets).
void snoopAllEnable();	Snoops ALL incoming packets.
void snoopAllDisable();	Disables the snoopAllEnable().
bool filterMAC(const Eaddr &MAC_address);	Returns 'true' if the MAC address is something the VMAC is interested in.
Eaddr getAddress();	Returns the VMAC's current MAC address.
void setAddress(const Eaddr &MAC_address);	Replaces the VMAC's current MAC address with a new address.
bool incomingPacketAvailable();	Returns true if there is incoming data available.

Public Method	Description
DynamicBufT popIncomingPacket();	Returns a copy of the accumulated incoming data, and clears its incoming-data-buffer.
void sendPacket(DynamicBufT& buf);	Sends a raw packet, no formatting is done on the packet.
void setIFG(int _IFG);	Sets the inter-frame-gap used.
void setMACtimeout (int _MACtimeout);	Sets a timeout on the MAC trying to send a packet.

CPeer Class Methods

Constructor:

The Constructor for CPeer requires a Virtual MAC instance as an argument.

```
CPeer(VMAC &_vmac);
```

The CPeer methods are discussed below

Table A.3 CPeer Class Methods

Public Method	Description
void init();	This initializes the CPeer instance, and registers this instance with the Virtual MAC (defined in the constructor). This also starts up two threads responsible for processing incoming and outgoing packets respectively.
void setMACAddress(Eaddr &addr);	Sets the Peer's MAC address. Note: this also updates the MAC address of the Virtual MAC defined in the constructor.
void setIPAddress(Ipaddr &addr);	Sets the Peer's IP address.
Eaddr lookup_MAC(IPaddr addr);	Looks up the IP address in the ARP table and returns a MAC address. Note: this creates a random entry if the entry does not exist.
void setDUTAddress(Eaddr &targetMAC, IPaddr &targetIP);	Sets the MAC address & IP address of the target (the Design Under Test). Due to the limitations of CPeer there can be only one target per instance.
void sendArpRequest(Eaddr &targetMAC, IPaddr &targetIP);	Sends an ARP request with the specified target MAC and IP address.

Public Method	Description
void addArpEntry(Eaddr &MACaddress, IPaddr &IPaddress);	Adds an IP <-> MAC address association in CPeer's Route-Table.
void waitSendPacket();	Waits until CPeer has sent an ETH layer packet.
void waitReceivePacket();	Waits until CPeer has received & processed an ETH layer packet.
void waitSendOrReceivePacket();	Waits until either CPeer has sent an ETH layer packet OR it has received & processed an ETH layer packet.

Connection Class

There are three classes used for managing connections:

- *ICMPConnection*
- *UDPConnection*
- TCPConnection

These classes are all derived from the *Connection* class. Following tables explains the methods related to each class

Table A.4 Connection Class Methods.

Public Method	Description
string name();	Returns a Socket handle name.

Public Method	Description
void orderPkt(OrderEnum field);	Specifies an Order constraint on the packet.
unsigned int readReceiveBuffer (unsigned int length, char *msg);	Reads up to 'length' bytes into a user-provided buffer 'msg' returns the actual number of bytes copied into 'msg'.
unsigned int readReceiveBufferByte (unsigned int index);	Reads the 'index' byte entry in the receive-buffer and returns 1 if successful, 0 if failed.
unsigned int getReceiveBufferSize();	Returns the size of the receive buffer.
void ping(int id, int seq, int len, char *message);	Sends a PING packet with a particular ID, sequence number, and a message of length 'len'.
void addIPOptions(unsigned long option);	Adds an IP option to the connection.
void clearIPOptions();	Clears all IP options.
void setTypeOfService(unsigned short value);	Sets the Type Of Service field in the IP Header.
void setIPid(unsigned short value);	Sets the ID field in the IP Header.
void setFragmentOffset(unsigned int value);	Sets the Fragment Offset field in the IP Header.
void setTimeToLive(unsigned short value);	Sets the Time-to-live in the IP Header.
void corruptClear();	Clears all Corruption Events.

Public Method	Description
void corruptIPVersion (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Version field in the IP Header.
void corruptIPHeaderLength (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Header Length field in the IP Header.
void corruptIPTotalLength (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Total Length field in the IP Header.
void corruptIPFragmentOffset (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Fragment Offset field in the IP Header.
void corruptIPProtocol (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Protocol field in the IP Header.
void corruptIPChecksum (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Checksum field in the IP Header.
void corruptIPSrc (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Source IP field in the IP Header.

Public Method	Description
void corruptIPDst (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Destination IP field in the IP Header.

ICMPConnection Class

ICMP Connections are created and managed through this class. It inherits all the public methods from the Connection class.

Constructor:

The Constructor must specify a CPeer and a destination IP address.

ICMPConnection(CPeer &, unsigned long dst);

Table A.5 ICMP Connection Class Methods

Public Method	Description
void sendRawData(int type, int len, char *message);	Sends a packet of type 'type', with a message of length 'len'.
void setType(unsigned char value);	Sets the Type field of outgoing packets.
void setCode(unsigned char value);	Sets the Code field of outgoing packets.
void setId(unsigned short value);	Sets the ID field of outgoing packets.

Public Method	Description
void setSeqNum(unsigned short value);	Sets the starting sequence number of outgoing packets.
unsigned short getNumReplies();	Returns the number of replies it has aggregated on this connection.
void corruptICMPCode (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the ICMP Code field. See the
void corruptICMPChecksum (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the ICMP Checksum field.

UDPConnection Class

UDP Connections are created and managed through this class. It inherits all the public methods from the Connection class.

Constructor:

The Constructor must specify a CPeer, a source port, a destination IP and a destination port.

UDPConnection(CPeer &, int srcPort, unsigned long dstIP, int dstPort);

Table A.6 UDP Connection Class Methods

Public Method	Description
void sendDataFile(char *filename);	Sends the contents of a file to the destination.
void dumpOutputData(char *filename);	Dumps the received data to a file.
void sendData(DynamicBufT &data);	Sends the contents of a buffer to the destination.
void receiveData(DynamicBufT &data);	Appends the received data to a buffer.
unsigned int getReceiveDataSize();	Returns the current size of the received data buffer.
void waitReceivePacket();	Waits for the next received packet.
unsigned int getSendDataPendingSize();	Returns the current size of the send data buffer that is pending.
void waitSendPacket();	Waits for the next packet to be sent out.
void write(int len, char *message);	Sends 'len' bytes of a message.
void corruptUDPHeaderLength (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Header Length field in the UDP Header.
void corruptUDPChecksum (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the UDP Checksum in the UDP Header.

TCPConnection Class

TCP Connections are created and managed through this class. It inherits all the public methods from the Connection class.

Constructor:

The Constructor requires a CPeer, a source port, a destination IP and a destination port.TCPConnection(CPeer &, int srcPort, unsigned long dstIP, int dstPort);

Table A.7 TCP Connection Class Methods

Public Method	**Description**
void sendDataFile(char *filename);	Sends the contents of a file to the destination.
void dumpOutputData(char *filename);	Dumps the received data to a file.
void sendData(DynamicBufT &data);	Sends the contents of a buffer to the destination.
void receiveData(DynamicBufT &data);	Appends the received data to a buffer.
unsigned int getReceiveDataSize();	Returns the current size of the received data buffer.
void waitReceivePacket();	Waits for the next received packet.
unsigned int getSendDataPendingSize();	Returns the current size of the send data buffer that is pending.

Public Method	**Description**
void waitSendPacket();	Waits for the next packet to be sent out.
void write(int len, char *message);	Sends 'len' bytes of a message.
void sendSyn();	Sends a SYN packet and updates CPeer's TCP state-machine.
void sendFin();	Sends a FIN packet and updates CPeer's TCP state-machine.
void reset();	Resets the TCP state-machine
unsigned int sendPacket (int flags, unsigned int urgent, bool sendEmptyPacket = false);	Explicitly sends a packet, with specified flags and urgent status. The *sendEmptyPacket* parameter is an optional parameter that when set to *true* forces a packet to be sent even if no payload can be sent. When false, a packet is *not* sent if no payload is available. For example, if the advertised-window-size is zero, or if there is no data left to send, then no payload can be sent. If set to *true* then a packet with the specified flags and urgent pointer will be sent anyway, but with empty payload. By default this parameter is set to '*false*'.
int setAutoSendFlags(int flags);	Sets the flags used when auto-sending packets, and returns the previous value.

Public Method	Description
int setAutoSendUrgent(int urgent);	Sets the urgent status used when auto-sending packets, returns the previous value.
bool enableAutoSend();	Enables the auto-sending of pending data. Returns the old enable status (true implies it was enabled).
bool disableAutoSend();	Disables the auto-sending of pending data. Returns the old enable status (true implies it was enabled).
void setSeqNum(unsigned int value);	Sets the Sequence Number. Use this after instantiating a TCPConnection to set the Initial Sequence Number.
void setAckNum(unsigned int value);	Sets the Acknowledge Number.
void setWindowSize(unsigned short value);	Sets the transmission window size.
void setWindowSize32(unsigned long value);	Sets the PEER transmission window size of 32 bits. It will take effect when Window scaling(TCP Options) is enabled.
void setSendSize(unsigned short value);	Sets the transmission send size.
void setTcpState(tcpStateEnum value);	Forces the TCP state-machine to a specific state. See below for a listing of the tcpStateEnum type.

Public Method	Description
void setTcpOptions(unsigned int value);	This function is DEPRECATED now. Instead individual function are added to set each TCP option
void setTCPOptEOL (unsigned short kind);	Sets the End of Option List (EOL) TCP Option
void setTCPOptNOP (unsigned short kind);	Sets the NO-Operation (NOP) TCP Option
void setTCPOptMSS (unsigned short kind, unsigned short len,	Sets the window MSS TCP Option in the Peer's TCP header
void setTCPOptWinScale (unsigned short kind, unsigned short len, unsigned short scale);	Sets the Window Scaling factor TCP options in the Peer's TCP header
void setTCPOptSACKPermitted (unsigned short kind, unsigned short len) ;	Sets the SACK Permitted TCP Option in the Peer's TCP header
void setTCPOptSACK (unsigned short kind, unsigned short len, unsigned short origin, unsigned short blksize);	Sets the SACK TCP Option (relative origin and Block Size) in the Peer's TCP header.
void setTCPOptTimeStamp (unsigned short kind, unsigned short len, unsigned long tsval, unsigned long tsechore);	Sets the TCP Timestamp Option in the Peer's TCP header

Public Method	Description
unsigned short getTCPOptMSS(void);	Returns the MSS value in incoming packet
unsigned short getTCPOptWinScale (void);	Returns the Window Scale Factor in incoming packet
unsigned long getWindowSize32(void);	Returns the current 32-bit window size received in an incoming packet.
Unsigned short getTCPOptSACKPermitted(void);	Returns the Sack permitted value in an incoming packet
unsigned short getTCPOptSACKLength(void);	Returns the length of the SACK Option
unsigned short getTCPOptSACKRelOrigin(void);	Returns the first relative Origin of the SACK Option
unsigned short getTCPOptSACKBlkSize(void);	Returns the first Block Size of the SACK Option
unsigned long getTCPOptTSval(void);	Returns the TS value from the received packet
unsigned long getTCPOptTSEchRe(void);	Returns the TS Echo Reply value from received packet

Public Method	Description
unsigned int getSeqNum();	Returns the current Sequence Number.
unsigned int getAckNum();	Returns the current Acknowledgement Number.
unsigned short getWindowSize();	Returns the current window size.
unsigned int getSendSize();	Returns the current send size.
tcpStateEnum getTcpState();	Returns the current TCP state-machine's state.
void waitTcpState(tcpStateEnum value, unsigned int timeout=10000000);	Wait for the TCP connection to reach a state. There is a default timeout of 10 seconds of simulation time to reach this state.
int getTcpCode();	Returns the current TCP flags field.

Public Method	Description
void sendTcpSegment(DynamicBufT &buf, int flag, int windowMss, int urgPoint);	This sends the contents of 'buf' in TCP Segments with a particular Send Size, flags and urgent pointer. Note: During this operation TCP auto-send is disabled for all TCP connections running on this CPeer instance.
void corruptTCPChecksum (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Checksum field in the TCP Header.
void corruptTCPOptions (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Options field in the TCP Header.
void corruptTCPSrcPort (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Source Port field in the TCP Header.
void corruptTCPDstPort (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Destination Port field in the TCP Header.
void corruptTCPSeqNum (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Sequence Number field in the TCP Header.

Public Method	**Description**
void corruptTCPAckNum (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Acknowledgement Number field in the TCP Header.
void corruptDuplicate (Corrupt_Type type, Corrupt_Random is_random, unsigned long value);	Corrupts the Duplicate Instance of a TCP Packet.

Appendix B

TOE Configuration Tasks

This appendix describes the tasks that are used to configure various modules of the TOE to a desired state. The modules include:

- PCI interface
- Network Stack (TCP, IP, ARP, Ethernet interface, internal memories)
- GMAC
- External memories

Task Name: *configure_system*

Objective: Configure the TOE module to a desired state.

Input: NSIPadd; // IP address of Network Stack

Called By: Testcase

Description: It calls various tasks that configure different modules of TOE. They are as follows:

init_pci - To Initialize the PCI interface

configure_GMAC - To configure MAC

init_ns_mem - To configure external memories of TOE

configure_etherif_reg - To configure TOE's Ethernet interface

configure_IP_reg - To configure TOE's IP module

configure_TCP - To configure TOE's TCP module

configure_ARP_reg - To configure TOE's ARP module

MAC Tasks

These tasks are used to control the GMII interface of the MAC model.

Task Name: EthernetRx

Objective: Testbench Recieves a packet from the TOE's GMII interface

Inputs: none

Output: buffer; // A buffer to store an Ethernet frame received from
// NS GMII Interface.
size; // Size of Ethernet frame in bytes.

Called By: alwaysBlock_IncomingPackets

Description: It calls a MAC_TB task *testbench_receive_packet*, which receives a frame byte-by-byte from the GMII interface and stores it in a buffer. It returns the length of the received frame in bytes.

Task Name: EthernetTx

Objective: Testbench sends an Ethernet frame into the MII interface

Inputs: packet_data; // Buffer to store an Ethernet frame
// **received** from peer.
size; // Size of Ethernet frame in bytes.

Called By: sendPacket

Description: It calls a MAC_TB task *testbench_transmit_packet_from_buffer*, which forwards the frame to GMII interface of MAC

Status Message Tasks

These tasks are used to:

- Identify the status messages generated by the TOE
- Read status message data from the processor's memory
- Clear the current status message from the queue and
- Watch for a status message

TOE TCP Tasks

These tasks are used to control the TCP module of the TOE.

Task Name: *open_client_socket*

Objective: Create a NS client socket to connect to a server

Input: src_ip_add_val // Network Stack IP Address

dest_ip_add_val // Peer IP Address

src_port_val // Network Stack Port number

dest_port_val // Peer port number

rx_start_add_val // Start address of host mem buffer

rx_buff_len // Size of host memory buffer

Output: socket_handle_value // NS socket ID

Called By: Testcase

Description: This task creates a TCP (client) socket in the TOE. For a client socket it disables slow-start, auto-port assignments and enables Keep-Alive, Nagle modes. It also programs the starting address of a buffer & buffer size in host memory and destination MAC address (48'hcafedeadbeef). Once the new

socket is committed a socket handle value is returned. The handle value is used by the various tasks to retrieve the control block associated with the socket. This task commands the TOE to generate a SYN packet.

Task Name: *partial_open_client_socket*

Objective: Create a TOE client socket to connect to a server without setting Host Memory

Input: src_ip_add_val // Network Stack IP Address

dest_ip_add_val // Peer IP Address

src_port_val // Network Stack Port number

dest_port_val // Peer port number

Output: socket_handle_value // NS socket ID

Called By: Testcase

Description: This task creates a TCP (client) socket in the TOE. For a client socket it disables slow-start, auto-port assignments and enables Keep-Alive, Nagle modes. It also programs a fixed destination MAC address of 48'hcafedeadbeef. This task is similar to *open_client_socket* except for the fact that it does not program the buffers in host memory.

Task Name: *open_server_socket*

Objective: Create a TOE server socket

Input: src_ip_add_val // Network Stack IP Address

src_port_val // Network Stack Port number

rx_start_add_val // Start addr of host mem buffer

rx_buff_len // Size of host memory buffer

Called By: Testcase

Description: This task creates a TCP server socket in the Network Stack. For a server socket it disables slow-start, auto-port assignments and enables Keep-Alive, Nagle modes. It also confirms that the server socket does not allocate a control block before receiving any connection.

Task Name: *verif_socket_est*

Objective: Create a NS server socket

Input: timeout // Timeout for socket established state

Output: cb_handle // NS Socket ID

Called By: Testcase

Description: This task waits for a socket to go to established state. A socket handle value is returned if the socket goes to established state. The simulation stops if the socket does not go to established state within a timeout period.

Task Name: *confirm_socket_commit*

Objective: Confirm that NS socket register values are set properly

Input: timeout // Timeout for socket commit

Output: cb_handle // NS Socket ID

Called By: *check_rx_dma_buf*, *req_tx*, and all the client/server open socket tasks

Description: This task checks the correctness of socket register values while creating a server socket. If commit_done is not set within a timeout period then the simulation stops with an error message. Finally commit_done is cleared if it is asserted due to successful socket creation.

Task Name: *open_udp_socket_data*

Objective: Create a NS UDP socket to connect to a server (with data passing)

Input: src_ip_add_val // Network Stack IP Address

dest_ip_add_val // Peer IP Address

src_port_val // Network Stack Port number

dest_port_val // Peer port number

rx_start_add_val // Start addr of host mem buffer

rx_buff_len // Size of host memory buffer

tx_data_len // Length of Data to be transmitted

checksum_en // Enable checksum

Called By: Testcase

Description: This task creates a UDP socket in the TOE. For a UDP socket it enables UDP checksum calculation and programs the starting address of a buffer & buffer size in host memory. Finally it transmits a desired size UDP datagram.

Task Name: *open_udp_socket*

Objective: Create a TOE UDP socket to connect to a server (without data passing)

Input: src_ip_add_val // Network Stack IP Address

dest_ip_add_val // Peer IP Address

src_port_val // Network Stack Port number

dest_port_val // Peer port number

rx_start_add_val // Start addr of host mem buffer

rx_buff_len // Size of host memory buffer

checksum_en // Enable checksum

Called By: Testcase

Description: This task creates a UDP (client) socket in the TOE. For a UDP socket it enables UDP checksum calculation and programs the starting address of a buffer & buffer size in host memory.

Task Name: *close_socket*

Objective: Close a connection (NS server or client socket)

Input: socket_handle_val // NS socket ID to be closed

tmt // timeout (it must be positive)

Called By: Testcase, *closeNSthenpeer*, *closepeerthenNS*

Description: This task issues a close socket command for the specified socket handle. It also clears the socket status registers

Task Name: *check_tcp_state*

Objective: Check the TCP_STATE of NS socket for a given socket handle

Input: socket_handle_val // NS socket ID

tcp_state // Expected TCP state

tmt // timeout

Called By: Testcase , *closepeerthenNS*

Description: For a specified socket's handle it checks if a socket goes to the expected TCP state within a timeout period.

Task Name: *set_socket_handle*

Objective: To indicate the NS socket to be accessed

Input: socket_handle_val // NS socket ID

timeout // timeout

Called By: Testcase and several other socket related tasks

Description: This task writes the socket handle value in the socket handle register and then waits for socket specific registers to upload the correct values. Essentially it waits for assertion of sck_reg_val bit within a timeout period.

Task Name: *clear_status*

Objective: Clear the NS socket specific registers

Called By: Testcase and several other socket related tasks

Description: This task clears all the socket specific registers by clearing close done, commit done and sck_reg_val bits

Task Name: *req_tx*

Objective: Request data from host memory be transmitted

Input: socket_handle_val // Socket handle owning data

tx_length_val // Length of data in bytes

tx_start_add_val // Starting address of buffer in host mem

tmt // timeout

Called By: Testcase and *req_tx_buf*

Description: For a specified socket handle this task configures the transfer length (in bytes) and start address of host buffer. Finally it issues a TCP transmit command.

Task Name: *check_tx_dma*

Objective: Check if TX DMA has occurred successfully

Input: socket_handle_value // Socket handle owning data

timeout

Called By: *req_tx_buf(a C++ method)*, Testcase

Description: This task checks the status message to confirm that TX DMA has occurred successfully within a timeout period.

Task Name: *check_rx_dma_buf*

Objective: Check if the received packet is DMA'ed into host memory

Input: socket_handle // Socket handle owning data

exp_rx_offset_pointer

rcv_len // Size of RX payload

buff_start_addr // Start address of RX payload

buff_len // Length of Receive buffer in host memory

tmt // Timeout

Called By: *peerSendData2NS,* Testcase

Description: This task checks the status message to confirm that RX DMA has occurred successfully within a timeout period. Once DMA is successful it compares the contents of host memory with payload transmitted by peer. The host buffer list is one of the input parameters of this task.

Task Name: *complete_open_server_socket*

Objective: Set up buffers in host memory after receiving a connection to a server socket

Input: socket_handle_val // NS Socket ID

rx_start_add_val // Start address of host memory buffer

rx_buff_len // Size of host memory buffer

tmt // timeout

Called By: Testcase

Description: This tasks sets up buffers in host memory after receiving a connection to a server socket.

References

[1] IEEE Standard 802.3, 1998 Edition, "IEEE standard for Local and metropolitan area networks—Common specifications," *Institute of Electrical and Electronics Engineers*

[2] Andrew S. Tanenbaum, "Computer Networks," *Prentice Hall PTR, February 2000*

[3] "The OSI Reference Model," http://www.rad.com/networks/1994/osi/osi.htm

[4] "OSI Reference Model Lecture," http://munshi.sonoma.edu/s96/bus420/osilecture.htm

[5] "The Basics - The internet Project," http://tgos.org/internet/basics.html

[6] W.Richard Stevens, "TCP/IP Illustrated, Volume1: The Protocols," *Addison Wesley Longman, Inc., 16th Printing, February 2000*

[7] Clark D. D., and Tennenhouse D. L., "Architectural Considerations for a New Generation of Protocols," *Proc. ACM SIGCOMM 90, Aug. 1990.*

[8] http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2816279-1,00.html

[9] Dittia Z. D., Cox, J. R. J., and Parulkar, G. M., "Design of the APIC: A high performance ATM host-network interface chip," *Proc. IEEE INFOCOM 95, April 1995*

[10] PCI Special Interest group, "PCI Local Bus Specification," *Revision 2.2, December 1995.*

[11] Michael J. S. Smith, "Application-Specific Integrated Circuits,"*Addison-WesleyLongman, Inc., 1997.*

[12] Prakash Rashinkar, Peter Paterson, Leena Singh, "System-on-a-chip Verification. Methodology and Techniques," *Kluwer Academic Publishers , Second printing 2001.*

[13] Brain W. Kernigham and Dennis M. Ritchie, "The C Programming Language," *Prentice-Hall, Inc., 1988*

[14] Bjarne Stroustrup, "The C++ Programming Language," *Addison-Wesley, Third Edition, 1997.*

[15] Kanakia H., and Cheriton D. R., "The VMP network adapter board (NAB): high performance network communication for multiprocessors," *Computer Communication Review*, 18, *no. 4, pp. 175-87, Aug, 1988. 0146-4833.*

[16] J. K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware?," *In Proceedings of the Summer 1990 USENIX Conference, pages 247-256, June 1990*

[17] Pasquale J., Polyzos G., Anderson E., and Kompela V., "A Digital Video-conferencing Experiment Using DECstation 5000 Workstation and an FDDI Network, Internal Report," *Department of Computer Science and Engineering, University of California, San Diego, CA,* 1992

[18] Sherali Zeadally, G. Gheorghiu, A. F. J. Levi, "Throughout optimization for multimedia applications over high speed networks," *HPN-1997, 101-114*

[19] Fall K. and Pasquale J, "Exploiting In-Kernel Data Paths to improve I/O Throughput and CPU Availability," in *Processdings of Usenix Winter Technical*

Conference., San Diego, CA, Jan. 1993

[20] Garcia D., Cowan J., Still G., Madison C., Bradley M., and Potter K., "Future I/O," *6th International Conference on Parallel Interconnects (PI'99) (Formerly Known as MPPOI), Anchorage, AK, USA, pp. 238-42, 1999.*

[21] Anderson D. C., Chase J. S., Gadde S., Gallatin A. J., Yocum K. G., and Feeley M. J., "Cheating the I/O bottleneck: Network Storage with Trapeze/Myrinet," *USENIX 1998 Annual Technical Conference, New Orleans, Louisiana, Jun. 1998*

[22] Arpaci Mutlu and Copeland J.A., "TCP performance over heterogeneous networks," *Conference on Local Computer Networks ; 2000; p.536-54*

[23] Chen R., "Using TCP/IP in a client/server environment," *CMG Proceedings ; 1995; v.2, p.739-744*

[24] Clark D. D., Davie B. S., Farber D. J., Gopal I. S., Kadaba B. K., Sincoskie W. David, Smith J. M., and Tennenhouse David L., "An overview of the AURORA gigabit testbed," *Proceedings - IEEE INFOCOM ; 1992; v.2, p.569-581*

[25] Jacobson, R. Braden, D. Borman, RFC1323: "TCP Extensions for High Performance", May 1992. Obtain via http://www.cis.ohiostate.edu/htbin/rfc/rfc1323.html

[26] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, RFC2018: "TCP Selective Acknowledgment Options", October 1996. Obtain via ftp://ftp.ee.lbl.gov/papers/rfc2018.txt