Aurora 64B/66B Bus Functional Model v1.4

User Guide

UG508 (v1.0) June 24, 2009

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/24/09	1.0	Initial Xilinx release.

Table of Contents

Schedule of Figures 5

Schedule of Tables 7

Preface: About This Guide

Guide Contents 9
Additional Resources 10
Conventions 10
Typographical 10
Online Document 11

Chapter 1: Introduction

Chapter 2: Architecture

Aurora Architecture Model 17
Aurora Protocol Engine (APE) 21
Protocol Conformance 32
Continuous Conformance Checkers 32
Automatic Compliance 33

Chapter 3: User Interface

Configuring Parameter Values 37
Setting File Names 39
Simple Packet Generation 39
Error Injection 41
A Sample IDF File 42
IDF File for Only Data without Flow Control 42
IDF File for Only User Flow Control Packets 42
IDF File for Data with User Flow Control 43
IDF File for Data with All Traffic 44

Chapter 4: Language Interface and Test Bench Integration

Verilog PLI Interface 49
System Calls and Verilog PLI Wrapper 49
$bfm_initialize 49
$bfm_execute 52
ABFM 64B/66B Integration with Verilog Simulation Environment 52
VHDL FLI/VHPI Interface 54
Foreign Subprograms and VHDL FLI/VHPI Wrapper 54
ABFM 64B/66B Integration with VHDL Simulation Environment 62

Chapter 5: Simulating with Aurora 64B/66B BFM

Simulation Procedure 65
Analyzing Results 69
Report File 72

Schedule of Figures

Chapter 1: Introduction

Figure 1-1: **ABFM 64B/66B Environment** 14
Figure 1-2: **ABFM 64B/66B Configurations** 15

Chapter 2: Architecture

Figure 2-1: **Multi-Lane Aurora Architecture Model** 18
Figure 2-2: **IDF Interface and APE Module Interaction** 19
Figure 2-3: **Modules in Aurora Architecture Model** 20
Figure 2-4: **Initialization Modules** 22
Figure 2-5: **Lane Striping Operation** 27
Figure 2-6: **Lane Destriping Operation** 27
Figure 2-7: **Scrambler** 30
Figure 2-8: **Block Alignment Logic** 31
Figure 2-9: **Descrambler** 31
Figure 2-10: **Automatic Compliance Flow Chart - Part 1** 35
Figure 2-11: **Automatic Compliance Flow Chart [Part 2]** 36

Chapter 3: User Interface

Chapter 4: Language Interface and Test Bench Integration

Chapter 5: Simulating with Aurora 64B/66B BFM

Schedule of Tables

Chapter 1: Introduction

Chapter 2: Architecture

Table 2-1: **Initialization State Machine** 23
Table 2-2: **Scheduling Algorithm in SA0 Mode** 23
Table 2-3: **Scheduling Algorithm in SA1 Mode** 24
Table 2-4: **TX Scheduler Output Signals** 25
Table 2-5: **Encoder I/O** 26
Table 2-6: **RX Distributor Functions** 28
Table 2-7: **Continuous Conformance Checkers** 32
Table 2-8: **Error Injection for Controlling State Transitions** 33

Chapter 3: User Interface

Table 3-1: **IDF Controlled Parameters in ABFM** 37
Table 3-2: **File id Used for Log Files** 39
Table 3-3: **Scheduling of Packets w.r.t. Time for IDF File for Data with All Traffic** . . . 44

Chapter 4: Language Interface and Test Bench Integration

Table 4-1: **Arguments of $bfm_initialize System Call** 49
Table 4-2: **Arguments of Foreign Subprogram bfm_initialize** 55
Table 4-3: **Arguments of Foreign Subprogram bfm_serial_bits** 58
Table 4-4: **Arguments of Foreign Subprogram bfm_execute** 60

Chapter 5: Simulating with Aurora 64B/66B BFM

Table 5-1: **Arguments of Script, run_test.pl** 67
Table 5-2: **Files in Results Directory** 70
Table 5-3: **Error Code Description** 73
Table 5-4: **File Pairs Used for Data Integrity Check** 74

About This Guide

Aurora 64B/66B bus functional model (ABFM 64B/66B) models the Aurora 64B/66B protocol as defined by *Aurora 64B/66B Protocol Specification v1.1* (SP011), www.xilinx.com/aurora. It is used for verifying an Aurora 64B/66B protocol implementation design which is referred to as device under test (DUT). This guide begins with an introduction to ABFM 64B/66B and describes its architecture in detail. ABFM 64B/66B is highly configurable both in the way it generates stimuli and in terms of functionalities and interfaces it supports and this guide describes the ways of configuring the ABFM 64B/66B in detail. The language interface provided by ABFM and the details of integrating and simulating with the ABFM 64B/66B verification environment are also discussed in this guide.

For more information on the Aurora 64B/66B Protocol, refer to *Aurora 64B/66B Protocol Specification v1.1* (SP011).

Guide Contents

This guide contains the following chapters:

- Preface, "About This Guide" introduces the organization and purpose of the user guide. A list of additional resources and the conventions used throughout the user guide is also documented in this chapter.
- Chapter 1, "Introduction" provides an overview of ABFM 64B/66B verification environment and lists the major features supported by ABFM 64B/66B v1.4
- Chapter 2, "Architecture" describes the overall implementation of Aurora 64B/66B BFM, providing details of Aurora 64B/66B protocol engine (APE) and the multi-gigabit transceiver (MGT) architecture model (MAM).
- Chapter 3, "User Interface" describes the format of user input files that are used for ABFM 64B/66B configuration, packet generation, and error injection.
- Chapter 4, "Language Interface and Test Bench Integration" describes Aurora 64B/66B BFM support for various language interfaces such as programmable language interface (PLI), foreign language interface (FLI), and VHDL procedural interface (VHPI). This chapter also describes the integration of language interface into simulation environment in detail.
- Chapter 5, "Simulating with Aurora 64B/66B BFM" describes the steps for simulating Aurora 64B/66B BFM and analyzing the simulation results.

Additional Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/literature.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

http://www.xilinx.com/support.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*`option_name`*] *`design_name`*
Braces { }	A list of items from which you must choose one or more	**`lowpwr =`**{**`on`**|**`off`**}
Vertical bar |	Separates items in a list of choices	**`lowpwr =`**{**`on`**|**`off`**}

Convention	Meaning or Use	Example

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis ...	Repetitive material that has been omitted	**`allow block`** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-5 FPGA User Guide.*
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

XILINX®

Chapter 1

Introduction

The Aurora 64B/66B bus functional model (ABFM 64B/66B) models behavior of the Aurora 64B/66B protocol and can be used to generate stimulus for, and to monitor response of an Aurora 64B/66B protocol implementation, which is referred to as the device under test (DUT). ABFM 64B/66B provides support for checking data integrity of incoming and outgoing data as well as validate DUT for Aurora 64B/66B protocol compliance. The ABFM 64B/66B is a clean room abstraction model of Aurora 64B/66B protocol that provides an increased confidence over using another ABFM 64B/66B design implementation to verify the DUT. Also, the use of abstraction model helps to perform verification earlier in the design process and reduces verification runtimes.

ABFM 64B/66B is designed such that it can be easily integrated into the verification environment of DUT. Easy integration is made possible through support for parameterization of design and protocol parameters, and test stimuli generation. Parameterization capability enables ABFM 64B/66B to test any implementation of DUT with little or no overhead.

The configurable parameters of ABFM 64B/66B are defined through either one of the following mechanisms:

- As a generic or parameter passed to the ABFM 64B/66B model through test bench: The parameters that are most likely to affect the interface of the DUT, such as the number of lanes and interface type (framing or streaming), are set through the test bench. As these parameters could possibly vary from one run of the simulation to another in order to test various interfaces of DUT, configuring them through test bench helps in easy integration. Users can define the DUT configuration in the top-level test bench as a set of generics or parameters, and pass it to both ABFM 64B/66B and the DUT.
- As a parameter defined through the input data format (IDF) file: The configuration parameters that are likely to affect the implementation but not the interface of DUT (such as the frequency with which CB is transmitted), and the configuration parameters that affect only the functionality of ABFM 64B/66B (such as the verbosity of a module) are defined through IDF file. For a given implementation of DUT, these parameters are expected to be static in nature, that is, they are less likely to vary from one run of simulation to the other. In addition, the IDF parameters also control test stimuli to the DUT such as packet generation and error injection.

For a given parameter, the configuration mechanism is fixed and can be configured only through the supported mechanism.

ABFM 64B/66B interfaces to the DUT through serial differential interface and communicate with DUT using programmable language interface (PLI) for Verilog environments. For VHDL, vendor supported interfaces are used: foreign language interface (FLI) for ModelSim and VHDL procedural interface (VHPI) for NCSim.

Figure 1-1 describes the ABFM 64B/66B environment used to verify the DUT implementation.

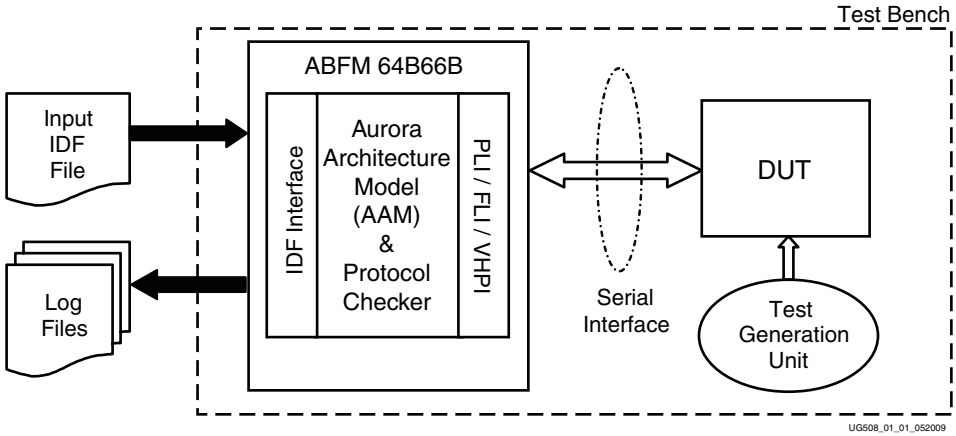

Figure 1-1: **ABFM 64B/66B Environment**

On the TX data path, the IDF reader processes the input IDF file at simulation time zero. The processed information is used by the Aurora architecture model to generate and schedule data and control packets in accordance with the Aurora 64B/66B protocol. The scheduled data is transmitted on the serial interface by the system calls. Transmitted data is logged in the transmit log files.

On the RX data path, PLI routines receive serial data from the DUT. The Aurora architecture model processes the received data to decode data and control packet information. The processed data is logged in receive log files. The transmit log files and receive log files are then compared to verify the functionality of the DUT.

The ABFM 64B/66B v1.4 provides the following features to verify the DUT implementation:

- Compliant to *Aurora 64B/66B Protocol Specification v1.1*
- Supports duplex or simplex modes of operation (Figure 1-2, page 15)
- Supports framing or streaming user interface
- Supports multi-lane configurations up to 24 lanes
- Supports both modes of NFC defined by protocol, NFC-Immediate and NFC-Completion modes
- Provides a simple user interface for parameter configuration, packet generation and error injection (Chapter 3, "User Interface")
- Provides support for protocol conformance checking for data and control packets
- Provides support for auto-compliance testing to verify channel initialization state machines ("Protocol Conformance," page 32)
- Easy to integrate into an existing verification environment designed to test DUT (Chapter 4, "Language Interface and Test Bench Integration")
- Provides support for easy debug: Configurable verbosity at module level for log files and distinct error codes to easily identify the type of violation of protocol conformance ("Analyzing Results," page 69)

The ABFM 64B/66B channel is made up of one or more lanes and can be configured in simplex or full duplex mode.

Figure 1-2 describes the various possible ways of interfacing the ABFM 64B/66B to the DUT in simplex and duplex modes of operation:

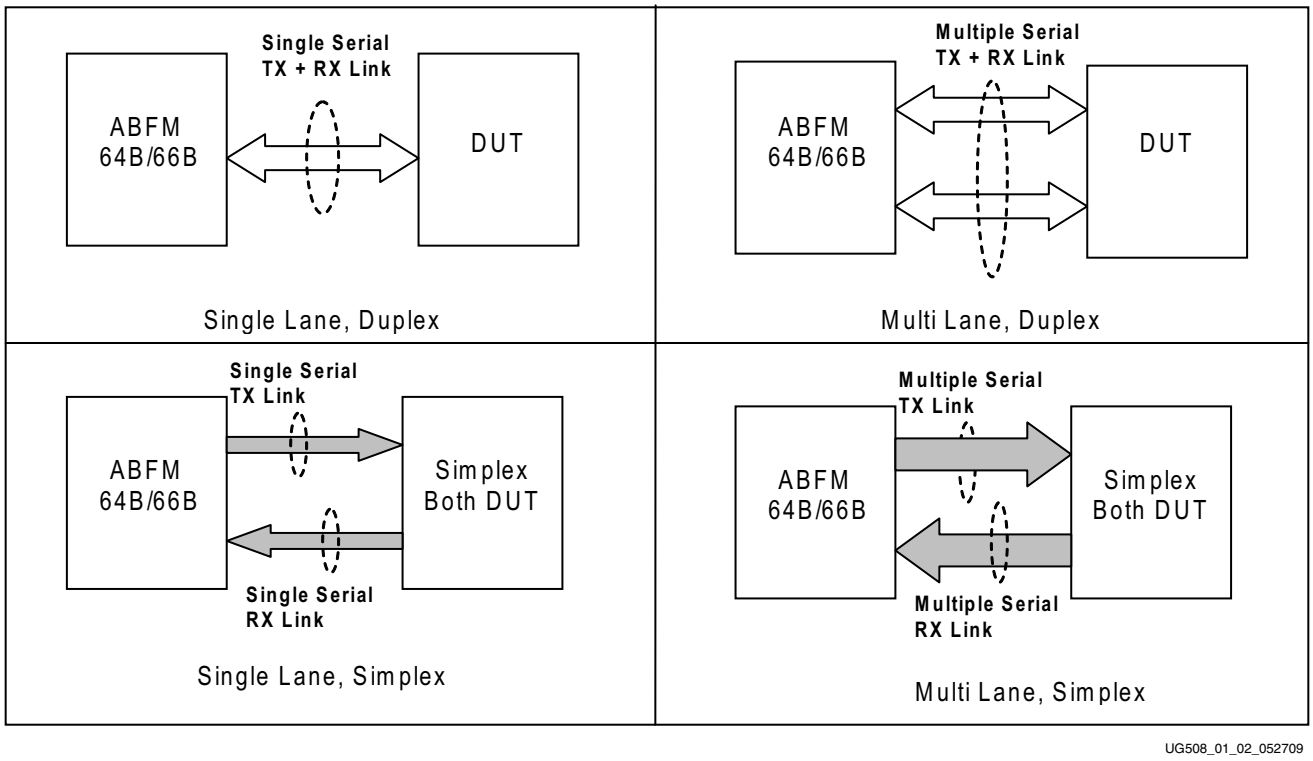

Figure 1-2: **ABFM 64B/66B Configurations**

XILINX®

Chapter 2

Architecture

The Aurora 64B/66B bus functional model (ABFM 64B/66B) is an abstraction model for the Aurora 64B/66B protocol and is used as a golden model to verify the protocol conformance of the DUT. It is implemented in C++ and distributed as a shared library that can be dynamically linked and simulated with the third-party simulators supporting PLI, FLI or VHPI interface. The architecture of ABFM 64B/66B can be broadly classified into:

- Aurora architecture model (AAM): Provides the functionality to fully emulate the Aurora 64B/66B protocol features. High-level test scenarios described in IDF format are translated into Aurora 64B/66B protocol compliant traffic and injected into DUT by means of PLI, FLI or VHPI interface. AAM also handles the error injection support.
- Checker module: Performs checking of traffic for overall data integrity and Aurora 64B/66B protocol compliance.

Aurora Architecture Model

Aurora architecture model emulates the legal behavior of an implementation of Aurora 64B/66B protocol as defined by specification. The rules defined by Aurora 64B/66B protocol can be broadly classified into the following categories:

- Channel control: Defines the link layer functionalities which cover rules for channel initialization, formation of single lane and multi-lane full duplex and simplex channel, frame delineation, flow control, and scheduling priorities for transmission of data and control information
- PCS/PMA functionality: Defines part of physical layer functionality such as rules for bit and byte ordering, data encoding and decoding, data scrambling and descrambling, 64B/66B gearbox operation, clock compensation, and channel bonding
- Electrical specification: Defines portion of physical layer dealing with signaling system such as threshold for signaling levels, minimum and maximum unit intervals, rise and fall times, maximum allowed jitter, and maximum allowed skew for Aurora serial link

The AAM models the channel control and the PCS/PMA functionalities. Based on the functionality being modeled AAM can be divided into two major modules:

- Aurora protocol engine (APE) which models the link layer channel control protocol definitions
- MGT (multi-gigabit transceiver) architecture model (MAM) which models the physical layer PCS/PMA functionality

The goal of AAM is only to verify the ABFM 64B/66B implementation, not the serializer/de-serializer operations in the real world or the implementation of the user interface in the DUT.

The Aurora protocol engine is unique for an Aurora 64B/66B channel, irrespective of the number of lanes while the MGT architecture model is unique to each Aurora 64B/66B lane. Figure 2-1 depicts the composition of AAM in a multi-lane configuration.

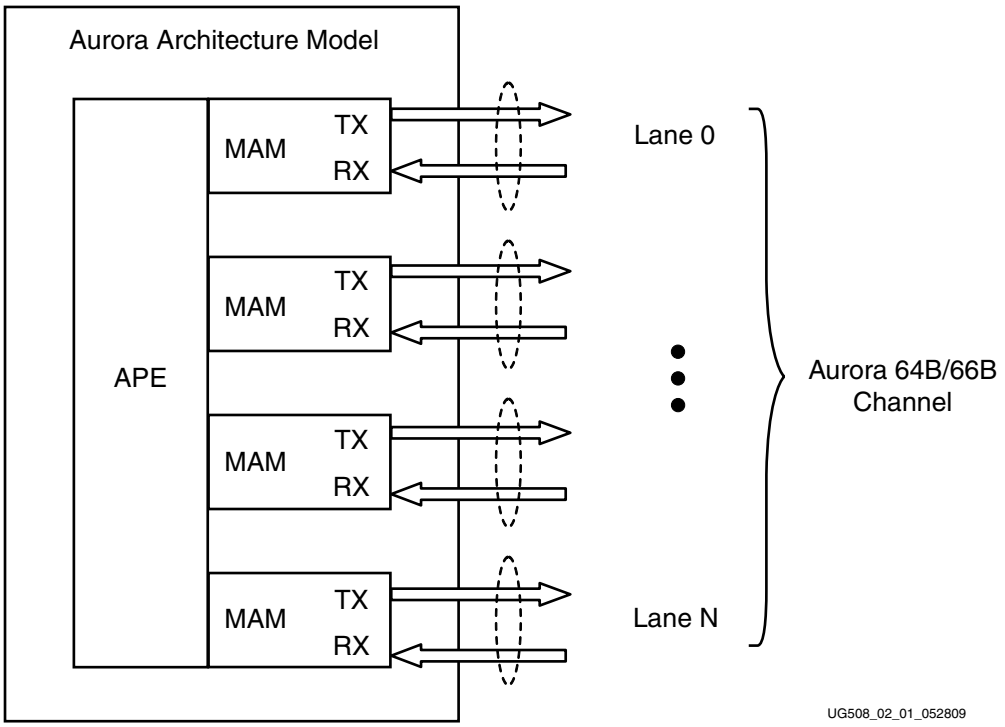

Figure 2-1: **Multi-Lane Aurora Architecture Model**

AAM is an abstract representation of Aurora 64B/66B protocol and is not meant to be cycle-accurate internally. Any timing-related behavior inherent in a protocol implementation (DUT) cannot be modeled in AAM. Therefore, the latencies as seen by the DUT implementation (such as pipelining data to meet timing) cannot be reproduced in ABFM 64B/66B simulation environment. However, the MAM provides support to skew lanes to model line characteristics.

Apart from emulating the legal behavior of Aurora 64B/66B protocol implementation, AAM is also responsible for generating test stimuli to verify the DUT. AAM uses file I/O technique to model user interface by which the user can apply test stimuli to send and receive different types of packets to and from ABFM 64B/66B. High-level test description given by input data format (IDF) files are read by IDF reader interface and converted into different types of packet requests supported by Aurora 64B/66B protocol. The APE module interacts with the IDF reader interface and translates the packet requests into Aurora 64B/66B compliant traffic and injects into DUT.

The IDF reader interface parses the user input IDF file at simulation time zero to read the high-level test description. It then converts the high-level test description into user packet data units (UPDU), USER K, native flow control (NFC), and user flow control (UFC) packet requests to be scheduled at different simulation time and queues them in packet FIFOs. There are four packet FIFOs internal to APE module, one for each type of packet: UPDU, USER K, NFC, and UFC.

Figure 2-2 describes the interaction between the IDF reader interface and the APE module:

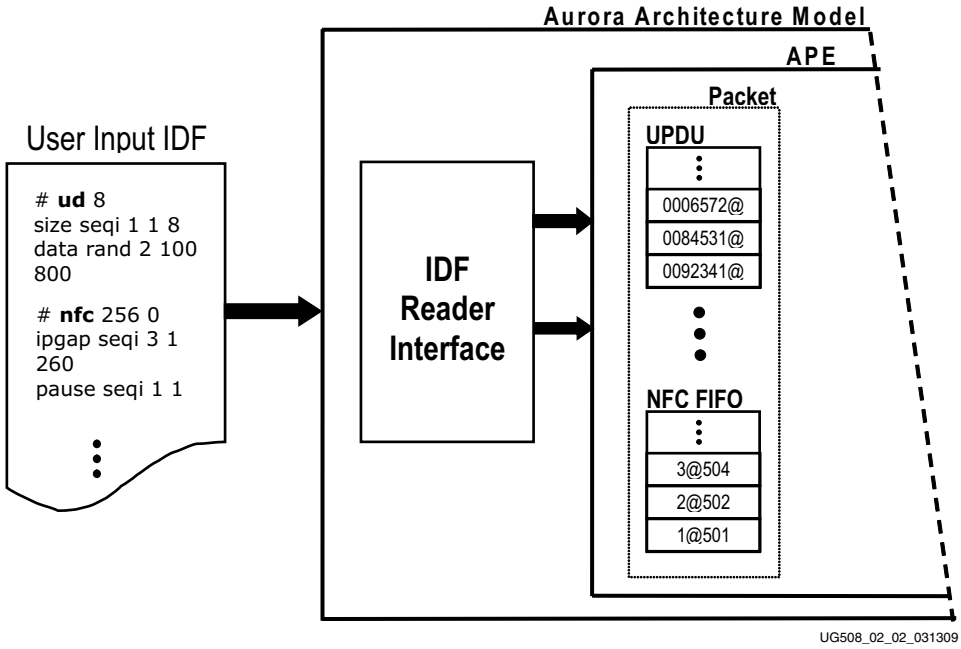

Figure 2-2: **IDF Interface and APE Module Interaction**

See Chapter 3, "User Interface" for more details on IDF file format used to describe test stimuli for generating different types of packets.

On the transmit side, at every clock cycle advancement in simulation time, APE module reads the packet FIFOs and schedules packet transmission in accordance with the priority rules defined by Aurora 64B/66B protocol. APE also logs the packets scheduled for transmission in transmit log files. The MAM modules perform the required PCS/PMA functionality on the packets scheduled by the APE to derive per lane serial stream. The serial stream is then injected into per lane serial differential lines by which the ABFM 64B/66B interfaces to the verification environment it is integrated with. The serial stream is injected into serial differential lines using Programmable Language Interface (PLI) routines for Verilog based verification environments. For VHDL based verification environments, vendor supported language interface is used: Foreign Language Interface (FLI) for ModelSim and VHDL Procedural Interface (VHPI) for NCSim.

On the receive side, the procedure is reversed. At every clock cycle advancement in simulation time, the PLI/FLI/VHPI routines are used to extract per lane serial stream from the verification environment. MAM modules then perform the required PCS/PMA functionality for the received serial stream followed by packet identification by the APE module. The packets are directly logged into receive log files. The transmit and receive log files are compared at the end of the simulation time to verify the correctness of DUT implementation.

The PLI/FLI/VHPI interface contains routines and functions that are used to move data to and from the serial differential lines of ABFM 64B/66B. These routines should be called using system calls from the verification environment ABFM 64B/66B is integrated with. For more details regarding these routines and the language interface, Chapter 4, "Language Interface and Test Bench Integration."

Figure 2-3 describes the modules in Aurora architecture model.

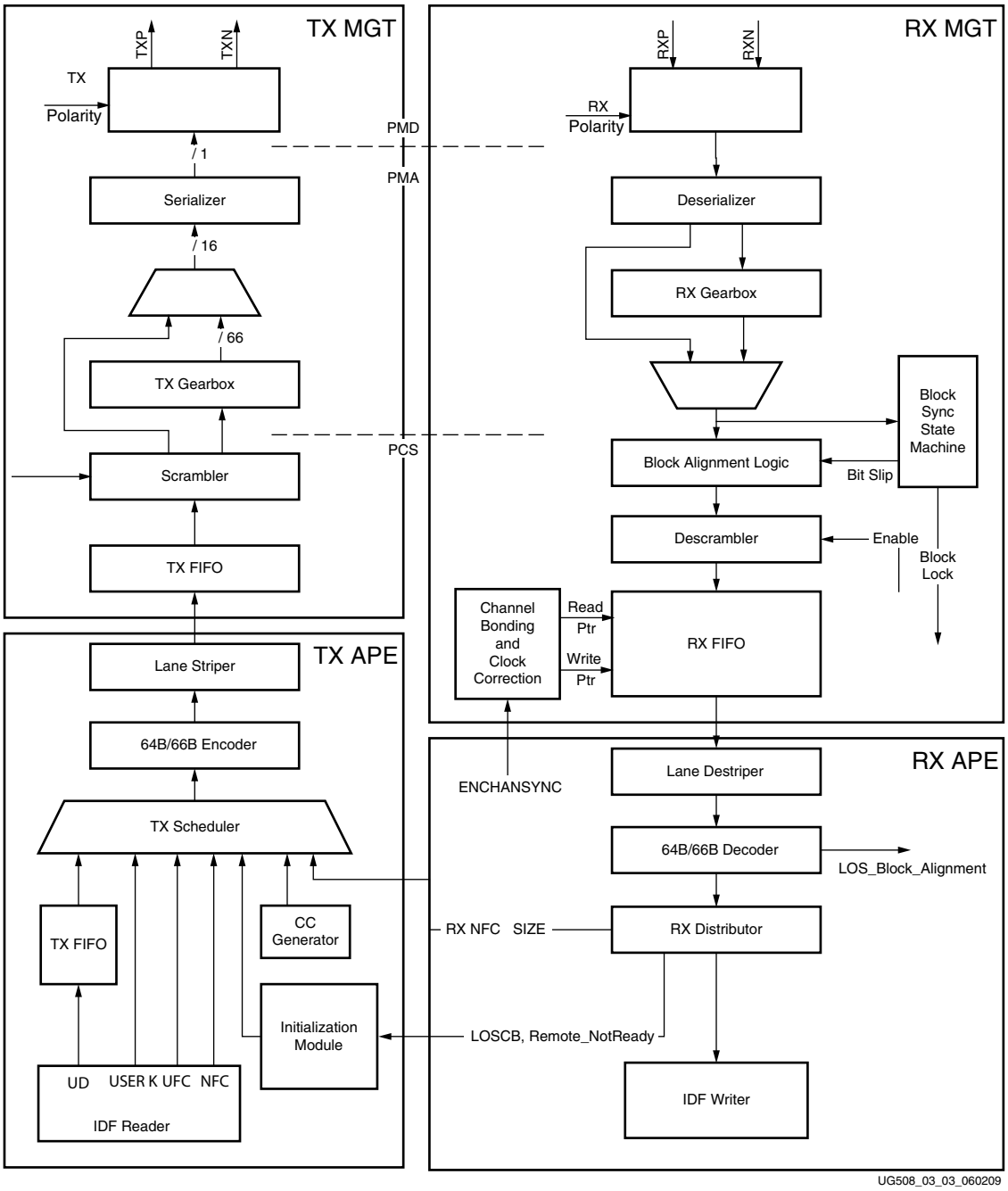

Figure 2-3: **Modules in Aurora Architecture Model**

Aurora Protocol Engine (APE)

Aurora protocol engine (APE) models the following link layer functions of Aurora 64B/66B protocol:

- Initializing channel ("Initialization Module")
- Scheduling of protocol defined control information along with user packet requests in accordance to the priority rules defined by the Aurora 64B/66B protocol ("TX Scheduler," page 23)
- Encoding of control information and packet data in Aurora 64B/66B block format ("Encoder," page 26)
- Striping blocks across different MAMs in multi-lane configuration ("Lane Striper," page 26)
- Destriping blocks received from different MAMs in multi-lane configuration ("Lane Destriper," page 27)
- Decoding of the received Aurora 64B/66B blocks ("Decoder," page 27)
- Performing protocol defined action for each control block received and logging data for each user packets received in file corresponding to the type of the user packet ("RX Distributor," page 27)

Initialization Module

Initialization module implements a state machine, which performs lane initialization and channel bonding. Based on its current state, initialization module requests TX scheduler to schedule various control blocks as defined by Aurora 64B/66B protocol. It also handles error conditions such as loss of block lock indication (LOS_BLOCK_LOCK) from MAM modules, loss of block alignment indication (LOS_Block_Alignment) from the decoder (APE module) and loss of channel bond indication (LOSCB) from RX distributor (APE module). If any lane loses block lock (Block_lock), the MAM module corresponding to that lane indicates the initialization module to restart lane initialization. Similarly, the decoder unit monitors the received blocks for valid block type field (BTF) and requests the initialization module to restart lane initialization in case of loss of block alignment due to excessive BTF errors. RX distributor module checks whether all lanes receive periodic CBs and indicates initialization module to restart channel bonding if it does not receive CBs simultaneously on all lanes.

Activities performed in each state of initialization module are shown in Figure 2-4, page 22.

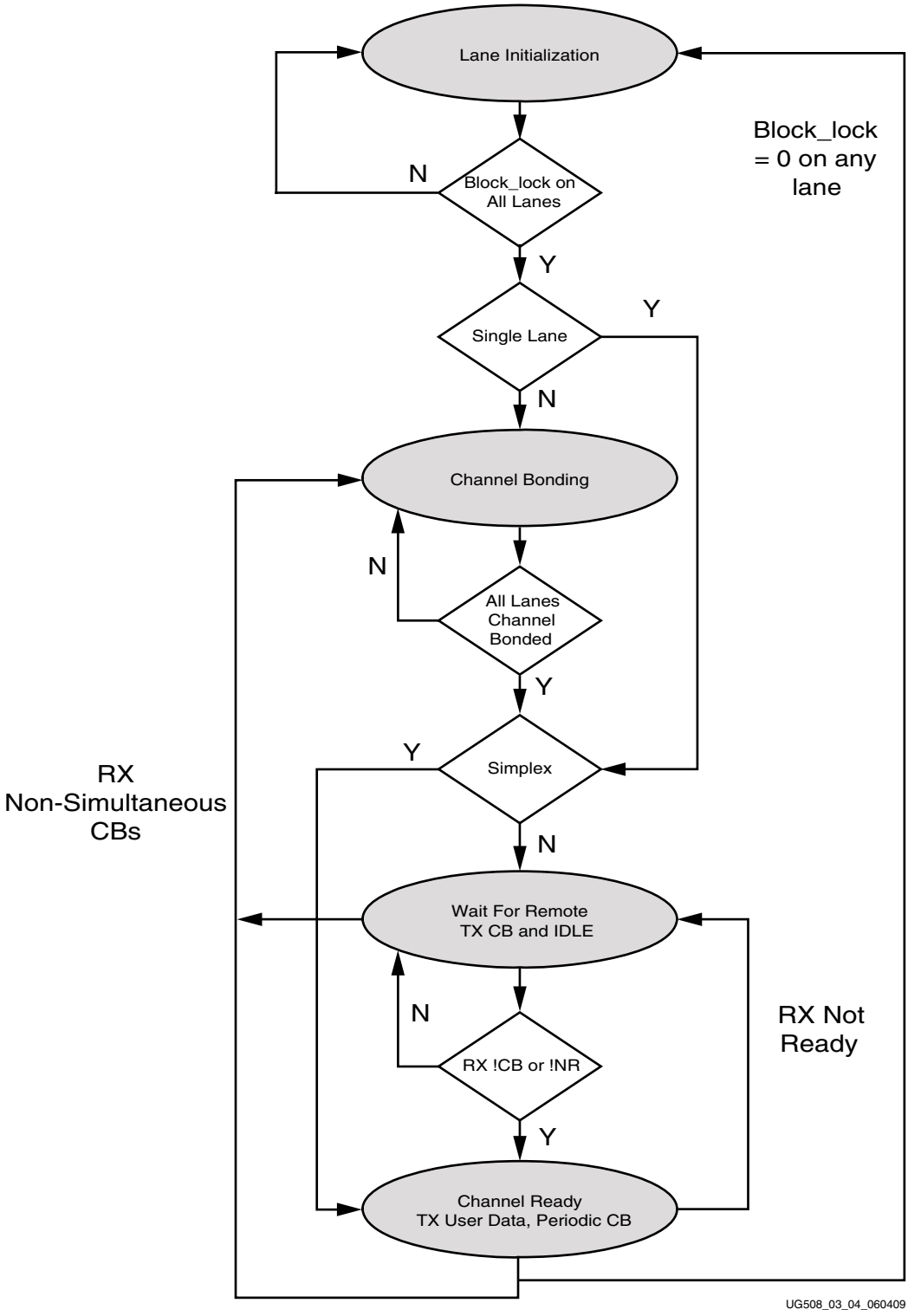

Figure 2-4: **Initialization Modules**

Table 2-1: **Initialization State Machine**

State	Transmitted Blocks
Lane Init	Repeatedly transmit <"nrinlaneinit"> NR + 1 CB block
Channel Bonding (only for multi-lane)	Repeatedly transmit <"nrinchbond"> NR + 1 CB block
Wait For Remote	Repeatedly transmit <"idleinwfr"> IDLE + 1 CB block
Channel Ready	Transmit any control block or user packet

TX Scheduler

TX scheduler schedules control blocks and user packets in accordance with the priority rules defined by Aurora 64B/66B protocol. The scheduling algorithm is a function of the strict alignment mode. It comes into play only when the initialization state machine is in Channel Ready state. In Lane Init, Channel Bonding, and Wait For Remote states, the transmitted control blocks are fixed. In Channel Ready state, on every clock cycle, the TX scheduler polls all the user packet requests (originated from the user input IDF file) and control block requests (originated from internal logic, for example periodic CC/CB transmit request) and uses the scheduling algorithm to decide what needs to be transmitted next.

Strict Alignment (SA)

Aurora 64B/66B protocol defines strict aligned (SA1) and non-strict aligned (SA0) modes of transmission. Strict alignment mode can be set for the ABFM 64B/66B at compile time through the input IDF file. The strict alignment information is encoded in a bit in the IDLE control block. By decoding the IDLE control blocks received, ABFM 64B/66B can determine the SA mode of its channel partner. If the SA mode of the channel partner is different, ABFM 64B/66B will change to the new SA mode.

Table 2-2 describes the scheduling algorithm in SA0 mode.

Table 2-2: **Scheduling Algorithm in SA0 Mode**

Control/Data Block	Condition for Transmission
Clock Compensation (CC)	Always transmitted periodically with highest priority
Not Ready (NR)	Transmitted in Lane Init and Channel Bonding states
Channel Bonding (CB)	Transmitted periodically in Channel Bonding, Wait For Remote, and Channel Ready states
Native Flow Control (NFC)	Transmitted in Channel Ready state when CC and CB are not pending. NR can never be pending since when NR needs to be transmitted, ABFM cannot be in Channel Ready state.
User Flow Control (UFC)	UFC is transmitted in Channel Ready state, when CC, CB, and NFC are not pending. It is used to start a UFC message.
USER K	USER K blocks are transmitted in Channel Ready state when CC, CB, NFC, and UFC requests are not pending.

Table 2-2: **Scheduling Algorithm in SA0 Mode *(Cont'd)***

Control/Data Block	Condition for Transmission
DATA	DATA block is used to transmit UPDU messages longer than 8 bytes and UFC messages. DATA block can be transmitted if CC, CB, NFC, UFC, USER K, and NFC PAUSE requests (Immediate mode) are not pending.
Separator (SEP/SEP7)	These blocks are used to mark end of UPDU message. Back-to-back SEP/SEP7 can be transmitted if UPDU messages are less than 8 bytes. They are not applicable for UFC message. Separator block can be transmitted if CC, CB, NFC, UFC, USER K, and NFC PAUSE requests (Immediate mode) are not pending.
IDLE	IDLE control block is transmitted in following cases: • In Wait For Remote state • In response to a received NFC request • If no other higher priority request is pending

Table 2-3 describes the scheduling algorithm in SA1 mode.

Table 2-3: **Scheduling Algorithm in SA1 Mode**

Control/Data Block	Condition for transmission
Clock Compensation (CC)	Always transmitted periodically with highest priority
Not Ready (NR)	Transmitted in Lane Init and Channel Bonding states
Channel Bonding (CB)	Transmitted periodically in Channel Bonding, Wait For Remote, and Channel Ready states
Native Flow Control (NFC)	NFC block is transmitted only on the last lane (least significant) when CC or CB requests are not pending.
User Flow Control (UFC)	UFC block is transmitted only on the last lane when CC, CB, and NFC requests are not pending.
USER K	USER K block is transmitted when CC, CB, NFC, and UFC requests are not pending.
DATA	Whenever DATA block of a packet is transmitted in a databeat, entire databeat should contain DATA blocks of that packet only. If SEP/SEP7 is transmitted in any databeat, then rest of the databeat should not contain any user data. This implies that it is not possible to transmit multiple UPDU packets or multiple UFC packets or a UPDU and UFC packet in the same databeat. However, it is permissible to transmit NFC or UFC control block on last code of this databeat. In a scenario where UFC or NFC request is pending, but DATA blocks of a packet do not occupy all the lanes in a databeat, it is possible to schedule DATA blocks in the most significant lanes and NFC or UFC block in the least significant lane.

Table 2-3: **Scheduling Algorithm in SA1 Mode *(Cont'd)***

Control/Data Block	Condition for transmission
Separator (SEP/SEP7)	SEP/SEP7 can be transmitted if CC, CB, NFC, UFC, USER K, and NFC PAUSE requests (Immediate mode) are not pending. The only exception to this rule is the case where UFC or NFC is transmitted on the last lane of the databeat and the UPDU packet can be accommodated in the remaining lanes.
IDLE	IDLE control block is transmitted in following cases: • In Wait For Remote state • In response to a received NFC request • If no other higher priority request is pending

After the TX Scheduler selects a Control/Data block for transmission, it generates outputs to the 64B/66B Encoder [see Table 2-4].

Table 2-4: **TX Scheduler Output Signals**

Block Selected for Transmission	Scheduler Output	Action Taken by 64B/66B Encoder
CC	Schedule_cc	Generates CC control block
NR	Schedule_nr	Generates NR control block
CB	Schedule_cb	Generates CB control block
NFC	Schedule_nfc Schedule_pause[8:0]	Generates IDLE control block with NFC=1, PAUSE = schedule_pause[7:0] XOFF = 1 if schedule_pause[8:0] = 0x1ff ON = 1 if schedule_pause[8:0] = 0x0
UFC	Schedule_ufc UFC_size[7:0]	Generates UFC control block, uses UFC_size and data as is
DATA	Schedule_data Data[63:0]	Encodes Sync Header (SH) = 01, uses data as is
SEP	Schedule_sep Data_count[7:0] Data[47:0]	Generates SEP control block, uses data_count and data as is
SEP7	Schedule_sep7 Data[55:0]	Generates SEP7, uses data as is
USER K	Schedule_user_k[8:0] Data[55:0]	Generates appropriate USER K control block, uses data as is
IDLE	Schedule_idle	Generates IDLE control block

Encoder

This module performs 64B/66B encoding on each block scheduled by transmit scheduler. Encoding is performed according to Table 2-5.

Table 2-5: **Encoder I/O**

TX Scheduler Output Signal	Encoder Output	
	SH	Encoder Data
Schedule_cc	10	78 80 00 00 00 00 00 00
Schedule_nr	10	78 20 00 00 00 00 00 00 - SA = 0 78 30 00 00 00 00 00 00 - SA = 1
Schedule_cb	10	78 40 00 00 00 00 00 00
Schedule_nfc Schedule_pause[8:0]	10	AA Pause 00 00 00 00 00 00
Schedule_ufc UFC_size[7:0]	10	2d Size 00 00 00 00 00 00
Schedule_data Data[63:0]	01	D[63:0]
Schedule_sep Data_count[7:0] Data[47:0]	10	1E Data_count Data
Schedule_sep7 Data[55:0]	10	E1 Data
Schedule_user_k[8:0] Data[55:0]	10	USER K-BTF Data
Schedule_idle	10	78 00 00 00 00 00 00 00 - SA = 0 78 10 00 00 00 00 00 00 - SA = 1

Lane Striper

Lane striper receives 64B/66B encoded data from encoder, stripes the data across the lanes, and transmits the data to the MGT model. Striping is meaningful only when there is more than one lane. Figure 2-5 describes lane striping operation for a 2-lane design. Lane striping ensures that received data is in correct order irrespective of data bus width between MGT and Aurora protocol engine.

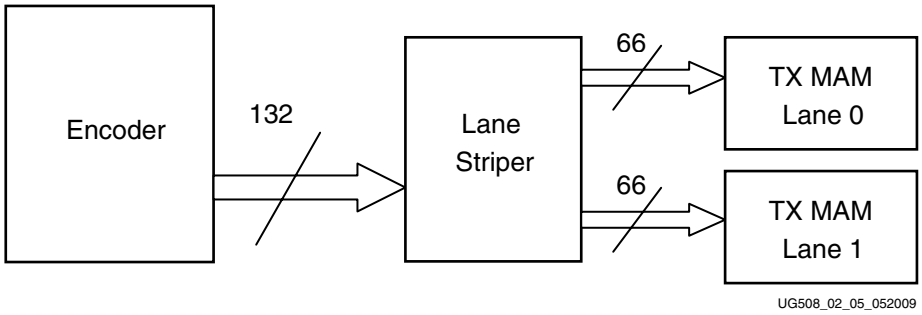

Figure 2-5: **Lane Striping Operation**

Lane Destriper

Lane destriper receives 64B/66B encoded data from each lane, destripes the data and sends it to the Decoder. Destriping is meaningful only when there is more than one lane. Figure 2-6 describes lane destriping operation for a 2-lane design.

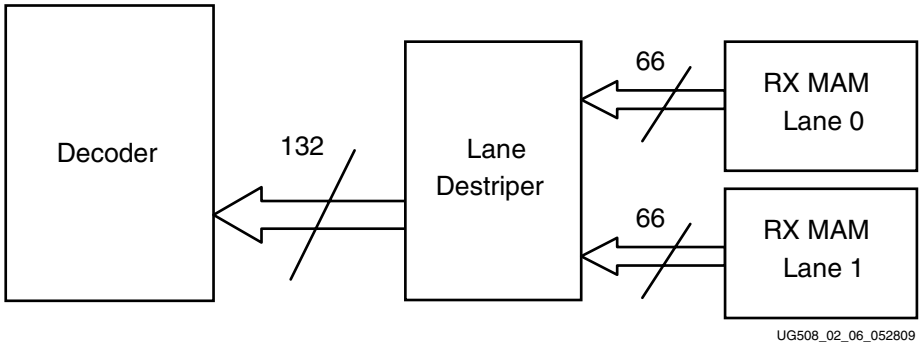

Figure 2-6: **Lane Destriping Operation**

Decoder

This module performs 64B/66B decoding on each block received from the lane destriper. Decoding is performed on each 66 bits block. If there are any decoding errors (SH or BTF errors), Decoder indicates the block to be invalid. Decoder maintains soft error count based on BTF errors (SH errors are monitored by block sync state machine) and if number of soft errors within a certain window exceed a certain threshold (defined by "soft_error_count_threshhold"), then it indicates loss of block lock (LOS_BLOCK_LOCK) to the initialization module.

RX Distributor

RX distributor discards control blocks and logs the user packets. The functions performed by RX distributor in each state are described in Table 2-6.

Table 2-6: **RX Distributor Functions**

Initialization Module State	Function Performed by RX Distributor
Lane Init	RX distributor is disabled in this state since block alignment is yet to be achieved.
Channel Bonding	RX distributor discards all the control/data blocks received in this state. This includes following: • Mix of CB and NR received when channel partner is in Channel Bonding state • Mix of CB and IDLE received when channel partner is in Wait For Remote state • Any UPDU, UFC, NFC transmitted by channel partner
Wait For Remote	RX distributor discards mix of CB and NR when channel partner is in Channel Bonding state. If CBs are not received on all lanes simultaneously, it asserts LOSCB signal. RX distributor discards CB and IDLE when channel partner is in Wait For Remote state. If CBs are not received on all lanes simultaneously, it asserts LOSCB signal. RX distributor does not discard any block when channel partner is in Channel Ready state.
Channel Ready	In this state RX distributor discards CC, CB, IDLE control blocks. If NR is received, it asserts Remote_NotReady signal. If CB is not received on all lanes simultaneously, it asserts LOSCB signal. If NFC is received, it outputs PAUSE and RXNFCREQ. If UFC is received, new UFC message reception is started, UFC size is also noted. All the DATA blocks are part of this UFC until remaining size is zero. If USER K is received, it is written to a file. If DATA block is received when remaining UFC size is zero, then it is treated as start of new UPDU packet. All subsequent DATA blocks are part of this UPDU until UFC, SEP or SEP7 control block is received. If SEP or SEP7 control block is received, it indicates end of packet. If UFC control block is received before SEP/SEP7 is received for the current UPDU packet in reception, then it is treated as an embedded UFC packet. In this case UPDU reception is resumed after entire UFC packet is received.

Checker

Checker monitors the decoded data for protocol conformance. Each protocol conformance check is associated with an error code. When the checker module detects an error, it prints the corresponding error code in the ABFM error log. For more information on error logs, refer to "Analyzing Results," page 69. The protocol conformance checks built into the checker and their corresponding error codes are described in "Protocol Conformance," page 32.

Error Injection

Error injection is a feature by which the normal protocol operation can be disrupted for checking the behavior of design in such conditions. This can be very useful to verify any deadlock conditions that the design may land into in case of an erratic behavior from the other peer. The data after getting scheduled by the scheduler it enters the error injection block if error injection is enabled. The error injection block corrupts the data and then passes onto the encoder block for transmission. In spite of the error injected at a particular instant the scheduler operation would continue normally after the error is injected. The error to be injected can be controlled by the IDF file at the zero time.

Various forms of error injection include converting a BTF to another or corrupting the value of sync header. The BNF format for error injection is described in Chapter 3, "User Interface."

MGT Architecture Model (MAM)

MGT architecture model (MAM) models the following physical layer functions of the MGT:

- Store data coming from lane striper module ("TX FIFO")
- Scrambling operation ("Scrambler")
- Parallel data-to-serial data conversion ("Serializer")
- Serial data-to-parallel data conversion ("Deserializer")
- Capture sync header bits ("Block Sync State Machine")
- Get block aligned data ("Block Alignment Logic")
- Descrambling operation ("Descrambler")
- Store block aligned data, perform channel bonding, and clock correction ("RX FIFO")

TX FIFO

TX FIFO stores data from lane striper and passes it to the scrambler module.

Scrambler

Scrambler receives 66-bit data from TX FIFO, scrambles 64-bit data leaving 2-bit sync header untouched, and drives 66-bit scrambled data to TX gearbox. Scrambling polynomial is $X^{58} + X^{39} + 1$. Figure 2-7 shows block diagram of scrambler.

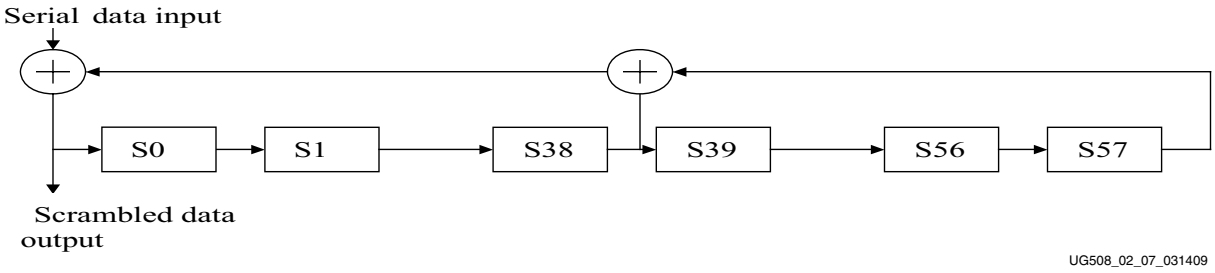

Figure 2-7: **Scrambler**

Serializer

Serializer models the PISO block. It receives 66-bit data and drives single-ended serial data. This data is then converted to differential output as function of TX polarity.

In ABFM, serializer is implemented in the PLI/FLI/VHPI routine.

Deserializer

Deserializer models the SIPO block. It receives serial data and drives 66-bit parallel data to the PCS layer. In ABFM deserializer is implemented in the PLI/FLI/VHPI routine.

Block Sync State Machine

Block sync state machine performs block alignment using 2-bit sync header field. While block alignment is being achieved, it generates Bit_slip output which is used to slide receive data window by one bit at a time. It generates Block_lock output to the Aurora protocol engine.

Block sync state machine stays active even during run time. Anytime after block lock is achieved, if it receives 16 sync header errors in 64 blocks, then it deasserts Block_lock output indicating loss of block alignment.

Block sync state machine will also indicate RX polarity depending upon whether it locks to SH = 10 or SH = 01.

Block Alignment Logic

Block alignment logic counts Bit_slip rising edges. Then it generates block aligned data from two consecutive 66-bit code words by selecting proper data blocks from the two code words (Figure 2-8).

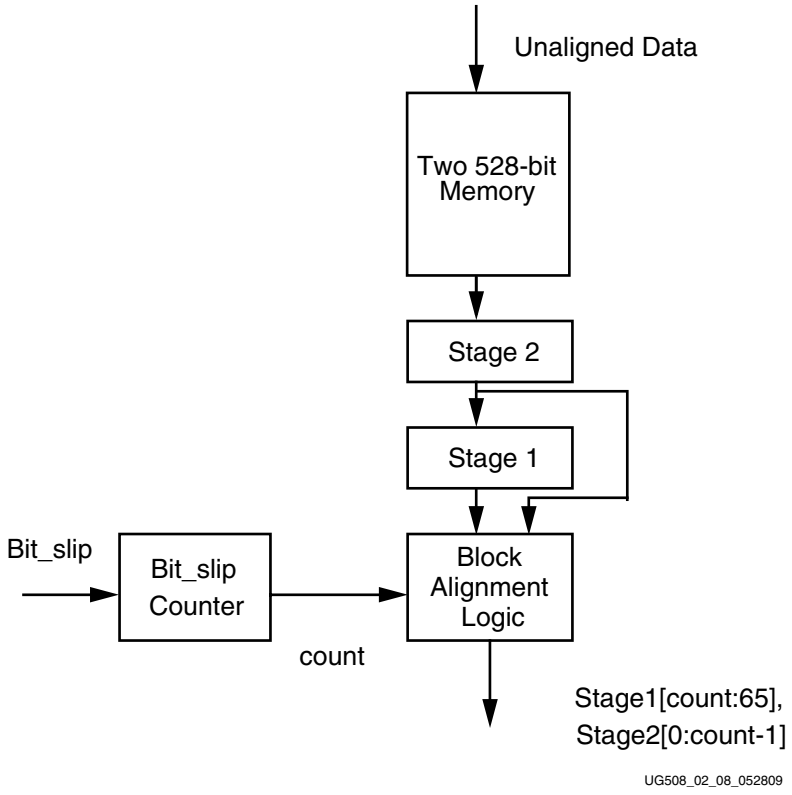

Figure 2-8: **Block Alignment Logic**

Descrambler

Descrambler receives 66-bit block aligned data, descrambles 64-bit data leaving 2-bit sync header untouched, and drives 66-bit unscrambled data to RX FIFO (Figure 2-9).

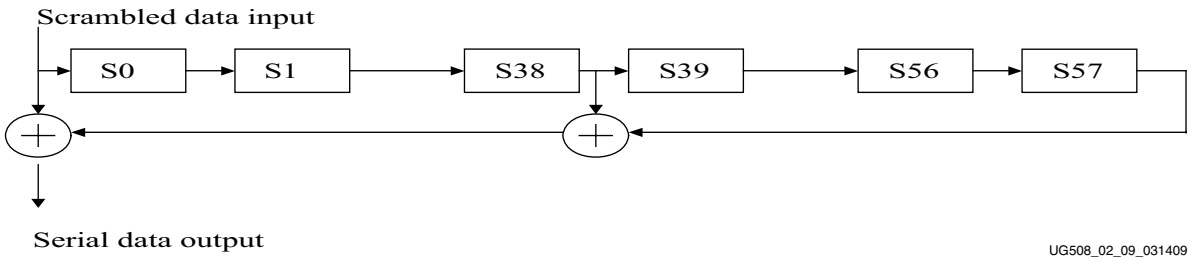

Figure 2-9: **Descrambler**

RX FIFO

RX FIFO stores descrambled data and passes it to the lane destriper module of RX APE.

Protocol Conformance

The checker module in ABFM implements several continuous conformance checks that check for conformance to Aurora 64B/66B protocol. Continuous conformance refers to checks performed by the ABFM continuously on the incoming data stream on the RX path. The ABFM cannot control the generation of data on the TX path of the DUT. It only monitors the data received on its RX path. However, if the DUT can be set into echo mode (wherein the RX path of the DUT is looped back into the TX path of the DUT), the ABFM can gain a good amount of control on the data received on its RX path. This provides enhanced testing if feasible.

The other way to test the DUT for protocol compliance is through automatic compliance feature. Automatic compliance mainly tests the initialization state machine of the DUT.

Continuous Conformance Checkers

Checker module implements continuous conformance checkers that monitor the incoming data on a per channel basis. When an error is detected, the corresponding error code is logged in the ABFM error log. Table 2-7 describes the checkers.

Table 2-7: **Continuous Conformance Checkers**

Checker	Checker Description
IDLE BTF Checker	CC, CB, NR, and IDLE control blocks share the common BTF 8'78. They are distinguished by the control bits (byte following BTF). If BTF matches with `8'h78`, but the control bits don't match with CC/CB/NR/IDLE, an error is flagged.
Soft Error Count Checker	Invalid SH bits and invalid BTF are treated as soft errors. A counter keeps a track of soft errors. If the count exceeds a threshold (defined by "soft_error_count_threshhold"), then an error is flagged and the LOS_BLOCK_LOCK signal is asserted.
UFC Checker	UFC checker flags an error when one of the following error conditions occur: • A new UFC header is received before the previous UFC packet was completely received (that is, the number of UFC data bytes received does not match with the UFC size specified in the UFC header for the previous UFC packet) • UFC header is not received in the last lane (least significant lane) in strict aligned mode (SA=1) • Multiple UFC packets are received in the same clock cycle in strict aligned mode • UFC and UPDU packets are received in the same clock cycle in strict aligned mode
NFC Checker	NFC checker flags an error when one of the following error conditions occur: • NFC received in simplex mode • Different (PAUSE or XOFF field is different) NFCs are received in the same clock cycle • In NFC-Immediate mode, if the time between transmission of NFC request and insertion of the first pause by the channel partner exceeds the time taken by the request originator to transmit 256 blocks

Table 2-7: **Continuous Conformance Checkers**

Checker	Checker Description
UPDU Checker	UPDU checker flags an error when one of the following error conditions occur: • Multiple UPDU packets received in the same clock cycle in strict aligned mode • SEP received in streaming mode
CC Checker	CC checker flags an error when one of the following error conditions occur: • CC is not received on all lanes simultaneously after channel bonding is complete • CC is not received with required length (defined by "cclen") • CC is not received with required frequency (defined by "ccfreq")
CB Checker	CB checker flags an error when one of the following error conditions occur: • CB is not received on all lanes simultaneously after channel bonding is complete • CB is not received with required length (defined by "cblen") • CB is not received with required frequency (defined by "cbfreq")
NR Checker	NR checker flags an error when NR is not received on all lanes simultaneously after channel bonding is done

Automatic Compliance

Automatic compliance feature provides a means to mainly test the initialization state machine of the DUT. Figure 2-10, page 35 and Figure 2-11, page 36 describe the automatic compliance flow-chart.

ABFM 64B/66B does not have direct control over the initialization state machine of the DUT. So, state transitions in the DUT can be triggered only by controlling the TX Data of ABFM 64B/66B. This is achieved by using the "Error Injection" feature. Table 2-8 describes how each state transition in the DUT is forced and the flowchart below describes the flow of states in detail.

Table 2-8: **Error Injection for Controlling State Transitions**

State Transmission	Error to be injected
From Lane Init to Channel Bonding	Valid 64 sync headers without any error injection
From Channel Bonding to Lane Init	16 invalid sync headers by error injection
From Channel Bonding to Wait For Remote	Combination of NR and CB blocks without error injection
From Wait For Remote back to Channel Bonding	Injecting an error by converting the IDLE blocks to a combination of NR and CB blocks. This forces the DUT out of Wait For Remote as CB is not received on all lanes.
From Wait For Remote back to Lane Init	Injecting an error by converting the valid sync headers of IDLE blocks to invalid sync headers.

Table 2-8: **Error Injection for Controlling State Transitions**

State Transmission	Error to be injected
From Wait For Remote to Channel Ready	By transmitting IDLEs without error injection
From Channel Ready to Wait For Remote	Injecting error by converting IDLEs to NRs
From Channel Ready to Channel Bonding	Injecting an error by converting the IDLE blocks to a combination of NR and CB blocks. This forces the DUT out of Wait For Remote as CB is not received on all lanes.
From Channel Ready to Lane Init	Injecting an error by converting the valid sync headers of IDLE blocks to invalid sync headers.

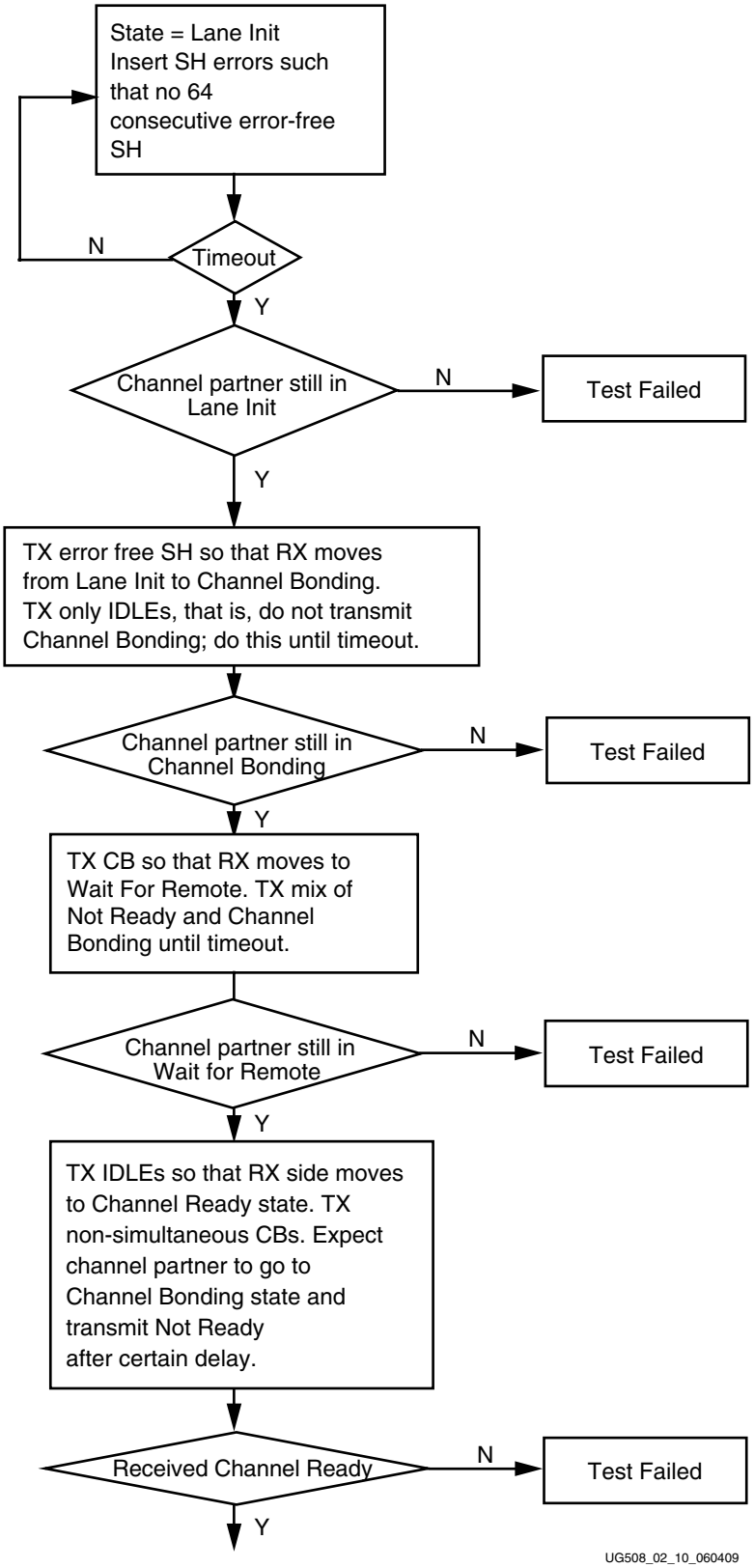

Figure 2-10: **Automatic Compliance Flow Chart - Part 1**

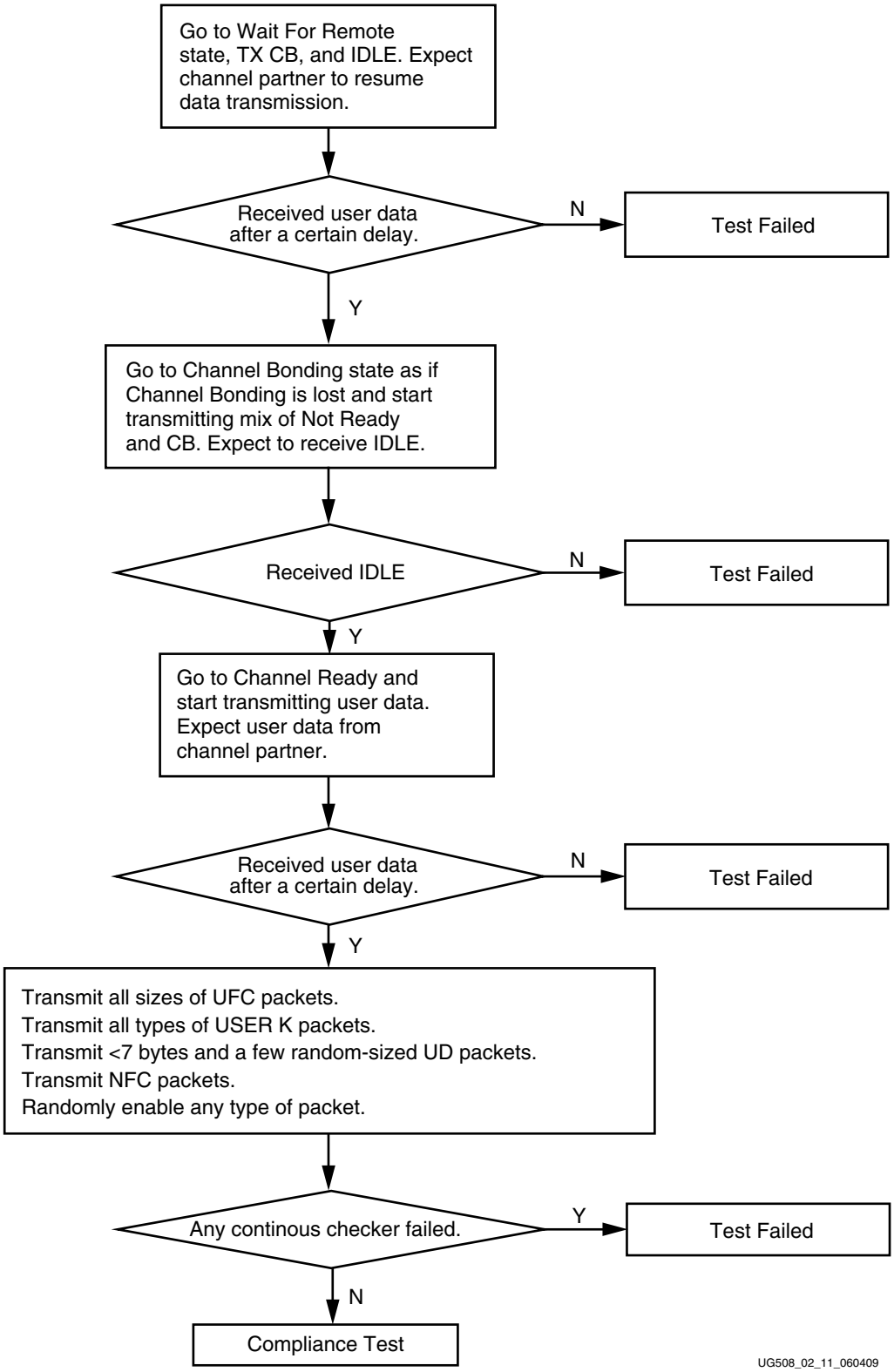

Figure 2-11: **Automatic Compliance Flow Chart [Part 2]**

Chapter 3

User Interface

Input data format (IDF) is a line-oriented file format that describes data frames and the controls for transmission of frames through the BFM.

This is an interface for the user to set or modify the values of various parameters in the BFM. The same interface is used to generate and control the stimulus to the BFM from the user. The packet format uses simple delimiters such as SOF/ EOF to describe packets.

Several options are provided to modify packets. The following section explains with examples in each.

- "Configuring Parameter Values"
- "Setting File Names," page 39
- "Simple Packet Generation," page 39
- "Error Injection," page 41

Configuring Parameter Values

The Aurora 64B/66B bus functional model (ABFM 64B/66B) provides parameters to be controlled by the user depending on their requirement (Table 3-1). Many of the ABFM 64B/66B configuration parameters and levels of verbosity for each module for easier debug can be set through these parameters.

This section describes the syntax for setting and modifying the ABFM 64B/66B configuration parameters.

Syntax: #setparam <parameter_name> <value>

Example:

```
# setparam ccfreq 10000
```

Table 3-1: **IDF Controlled Parameters in ABFM**

Parameter Name	Valid Values	Default Value	Description
samode	0,1	1	0 - SA0; 1 - SA1
maxerrorcount	Integer < 64	32	Determines the soft error count
ccfreq	Integer > 0	10000	Determines the frequency with which CC is transmitted.
cclen	Integer > 0	10000	Determines CC length. Actual CC period = ccfreq + cclen

Table 3-1: **IDF Controlled Parameters in ABFM *(Cont'd)***

Parameter Name	Valid Values	Default Value	Description
cbfreq	Integer > 0	10000	Determines the frequency with which CB is transmitted when channel is ready
cblen	Integer > 0	10000	Determines CB length. Actual CB period = cbfreq + cblen
nrinlaneinit	Integer > 0	4	Determines number of NR IDLE blocks transmitted in NR IDLE + CB pattern in the Lane Init state
nrinchbond	Integer > 0	4	Determines number of NR IDLE blocks transmitted in NR IDLE + CB pattern in the Lane Init state
idleinwfr	Integer > 0	4	Determines number of IDLE blocks transmitted in IDLE + CB pattern in Wait For Remote state
enableperiodiccb	0,1	1	1: Enables periodic CB transmission in Channel Ready state 0: Disable
simplex_li_ctr	Integer > 0	50	Determines the number of cycles for which SIMPLEX BFM initialization state machine should wait in the Lane Init state before moving to the next state
simplex_cb_ctr	Integer > 0	50	Determines the number of cycles for which SIMPLEX BFM initialization state machine should wait in the Channel Bonding state before moving to the next state
ENCODER	0-4	0	Determines verbosity of the encoder
STRIPER	0-4	0	Determines verbosity of the striper
DECODER	0-4	0	Determines verbosity of the decoder
DESTRIPER	0-4	0	Determines verbosity of the destriper
TX_SCHEDULER	0-4	0	Determines verbosity of the tx scheduler
PKT_SCHEDULER	0-4	0	Determines verbosity of the rx scheduler
RX_DISTRIBUTOR	0-4	0	Determines verbosity of the rx distributor
CHECKER	0-4	0	Determines verbosity of the checker
INIT_SM	0-4	0	Determines verbosity of the init state machine
APE	0-4	0	Determines verbosity of the ape
TX_MGT	0-4	0	Determines verbosity of the tx mgt
SCRAMBLER	0-4	0	Determines verbosity of the scrambler
RX_MGT	0-4	0	Determines verbosity of the rx mgt
DESCRAMBLER	0-4	0	Determines verbosity of the descrambler
ABFM	0-4	0	Determines verbosity of the ABFM

Setting File Names

This section describes the syntax for setting to set the file names that are written by the ABFM 64B/66B. There is no default option for the filenames and hence they must be defined in the IDF file.

Syntax: #setfile <file_id> <filename>

Example:

```
# setfile filetxufc "bfm0_tx_ufc_frames.dat"
```

Table 3-2: **File id Used for Log Files**

File_id	Function
Filetxufc	Transmitted UFC packets are written to this file
Filerxufc	Received UFC packets are written to this file
Filetxuserk	Transmitted USER K packets are written to this file
Filerxuserk	Received USER K packets are written to this file
Filetxpdu	Transmitted PDU packets are written to this file
Filerxpdu	Received PDU packets are written to this file

Simple Packet Generation

This section describes the syntax for generating the four different packets defined by the protocol, namely,

- "ud"(User Data/Payload Data Unit(PDU))
- "ufc" (User Flow Control)
- "nfc" (Native Flow Control)
- "userk" (USER K blocks)

"ud":

```
# ud <no_of_packets>
size <function>
data <function>
ipgap <function>
pause <function> duration <function>
```

"ufc":

```
# ufc <no_of_packets>
size <function>
data <function>
ipgap <function>
pause <function> duration <function>
```

Note: 1 <= size <= 256

"nfc":

```
# nfc <no_of_packets> <XOFF>
ipgap <function>
pause <function>

<XOFF> : 0,1
```

Note: pause < 256

"userk":

```
# userk <ktype> <no_of_packets>
data <function>
ipgap <function>

<ktype> : 0-8
```

<function>:

seqi min_val delta max_val

seqd max_val delta min_val

seqm min_val delta max_val

rand min_val seed max_val

const fixed_val

"ipgap" sets the inter-packet gap.

"pause" and "duration" set intra-packet gap

EXAMPLES:

- For sending 20 back-to-back UD packets of size 5000 bytes with random data:

```
# ud 20
size const 5000
data rand 2 10 800
ipgap const 0
pause const 0 duration const 0
```

- For sending 256 UFC packets ranging in size 1 to 256 bytes with random data while pausing for a random number of cycles between the packets:

```
# ufc 256
size seqi 1 1 256
data rand 1 10 1000
ipgap rand 20 10 50
pause const 0 duration const 0
```

For sending 256 NFC packets with decremental pause values while pausing for a random number of cycles between the packets:

```
# nfc 256 0
ipgap seqi 3 1 260
pause seqd 256 1 1
```

For sending seven back-to-back USER K packets of type 1:

```
# userk 7 1
data const 20
ipgap const 0
```

Error Injection

This section describes the syntax for using the error injection feature to inject errors into the TX stream of the BFM. There are three types of error injection namely:

- Symbol corruption
- Symbol deletion
- Symbol conversion.

Syntax:

```
# corrupt <error_parameter> [lane <integer>] [<actiontime>]
# delete <error_parameter> [lane <integer>] [<actiontime>]
# convert <error_from_parameter> <error_to_parameter> [lane <integer>]
[<actiontime>]

<error_parameter> | <error_from_parameter> | <error_to_parameter>:
sh
data_sh
ctrl_sh
btf
btf_cb
btf_cc
btf_nr
btf_nfc
btf_ufc
btf_sep
btf_sep7
btf_idle
btf_userk0
btf_userk1
btf_userk2
btf_userk3
btf_userk4
btf_userk5
btf_userk6
btf_userk7
btf_userk8

<actiontime>:
start <start_val> stop <stop_val>
  -- Execute from time=start_val to time=stop_val
start <on_period> period <off_period>
  -- Execute periodically
start <start_val> count <count_val>
  -- Execute from time=start_val to time=start_val+count_val
```

EXAMPLES:

In the following examples, time refers to ABFM global_time which is a count of the ABFM parallel clock cycles.

(Parallel clock period = serial clock period * 66 bits)

- For corrupting sync header bits on lane0 starting at time=340 and stopping at time=343:

  ```
  # corrupt sh lane 0 start 340 stop 343
  ```

- For deleting 200 regular IDLE blocks starting at time=100:

  ```
  # delete btf_idle start 100 count 200
  ```

For converting all regular IDLE blocks into NR IDLE blocks on lane0 starting at time=500 and stopping at time=507:

```
# convert btf_idle btf_nr lane 0 start 500 stop 507
```

A Sample IDF File

Here are few IDF files that are explained in detail.

IDF File for Only Data without Flow Control

```
# setparam ccfreq 10000
# setfile filetxufc "bfm0_tx_ufc_frames.dat"
# setfile filerxufc "bfm0_rx_ufc_frames.dat"
# setfile filetxpdu "bfm0_tx_pdu_frames.dat"
# setfile filerxpdu "bfm0_rx_pdu_frames.dat"
# setfile filetxuserk "bfm0_tx_userk_frames.dat"
# setfile filerxuserk "bfm0_rx_userk_frames.dat"

% Start of Test

% Send 8 UD packets
# ud 8
size const 100
data seqi 2 1 255
ipgap const rand 4 2 10
pause const 0 duration const 0

% End of Test
```

The file starts with setting parameters and then file names to log the packets both on transmit and receive interface.

1. The parameter 'ccfreq' sets the occurrence of clock correction sequence to every 10,000 cycles.
2. The IDF file generates eight user data packets of constant size 100 bytes each.
3. The first byte of the first packet is "02" and is incremented by 1 to get every other subsequent byte.
4. There are random number of IDLEs inserted in between the packets with the number of IDLEs ranging from 4 to 10. There can be few more IDLEs inserted based on NFC packets received from the other BFM.
5. User data packets start from 501 cycles onwards without any other packet being present.

IDF File for Only User Flow Control Packets

```
# setparam ccfreq 10000
# setfile filetxufc "bfm0_tx_ufc_frames.dat"
# setfile filerxufc "bfm0_rx_ufc_frames.dat"
# setfile filetxpdu "bfm0_tx_pdu_frames.dat"
# setfile filerxpdu "bfm0_rx_pdu_frames.dat"
# setfile filetxuserk "bfm0_tx_userk_frames.dat"
# setfile filerxuserk "bfm0_rx_userk_frames.dat"

% Start of Test
```

```
% Send 8 UD packets
# ufc 8
size seqi 8 8 64
data rand 1 10 256
ipgap const 20
pause const 0 duration const 0

% End of Test
```

1. Eight user flow control packets will be generated each with incrementing size and random data.
2. There will be 20 IDLE cycles between the successive packets.
3. The UFC packets gets scheduled from 501 cycle onwards.

IDF File for Data with User Flow Control

```
# setparam ccfreq 10000
# setfile filetxufc "bfm0_tx_ufc_frames.dat"
# setfile filerxufc "bfm0_rx_ufc_frames.dat"
# setfile filetxpdu "bfm0_tx_pdu_frames.dat"
# setfile filerxpdu "bfm0_rx_pdu_frames.dat"
# setfile filetxuserk "bfm0_tx_userk_frames.dat"
# setfile filerxuserk "bfm0_rx_userk_frames.dat"

% Start of Test

% Send 2 UD packets
# ud 2
size const 100
data seqi 2 1 255
ipgap const rand 4 2 10
pause const 0 duration const 0

% Send 2 UFC packets
# ufc 2
size const 8
data rand 1 10 256
ipgap const 20
pause const 0 duration const 0

% End of Test
```

1. Here there are two user flow control packets to be sent along with user data packets. First the user data and user flow control packets gets scheduled simultaneously at 501 cycle.
2. Since user flow control gets priority, the ufc packet of 8 bytes with random data is first sent out.
3. The user data packets then follow this user flow control packet.
4. After 20 cycles another user flow control packet interrupts the actual user data packet.

IDF File for Data with All Traffic

```
# setfile filetxufc "bfm1_tx_ufc_frames.dat"
# setfile filerxufc "bfm1_rx_ufc_frames.dat"
# setfile filetxpdu "bfm1_tx_pdu_frames.dat"
# setfile filerxpdu "bfm1_rx_pdu_frames.dat"
# setfile filetxuserk "bfm1_tx_userk_frames.dat"
# setfile filerxuserk "bfm1_rx_userk_frames.dat"

% Start of Test

 # userk 0 1
 data seqi 2 8 80
 ipgap const 0

 # userk 1 1
 data rand 4 8 80
 ipgap const 18

 # nfc 1 0
 ipgap const 0
 pause const 25

 # nfc 2 0
 ipgap const 9
 pause const 25

 # ud 2
 size const 64
 data seqi 2 1 255
 ipgap const  2
 pause const 0 duration const 0

 # ufc 2
 size seqi 8 8 24
 data seqi 1 10 256
 ipgap const 9
 pause const 0 duration const 0
```

Table 3-3, page 44 describes how the request is resolved and the corresponding data transmission for the above IDF file.

Table 3-3: **Scheduling of Packets w.r.t. Time for IDF File for Data with All Traffic**

SCHEDULING TIME/ PACKETS	UPDU	UFC	NFC	USER K	RESULT	Remarks
500						
501	UD1-1	UFC1-H	NFC1-H	USER K 0	NFC1-H	NFC has highest priority
502	UD1-2	UFC1-D	IDLE	IDLE	UFC1-H	UFC has next highest priority
503	UD1-3	IDLE	IDLE	IDLE	UFC1-D	
504	UD1-4	IDLE	IDLE	IDLE	USER K 0	USER K has more priority than user data

Table 3-3: **Scheduling of Packets w.r.t. Time for IDF File for Data with All Traffic *(Cont'd)***

SCHEDULING TIME/ PACKETS	UPDU	UFC	NFC	USER K	RESULT	Remarks
505	UD1-5	IDLE	IDLE	IDLE	UD1-1	First user data packet starts here
506	UD1-6	IDLE	IDLE	IDLE	UD1-2	
507	UD1-7	IDLE	IDLE	IDLE	UD1-3	
508	UD1-8	IDLE	IDLE	IDLE	UD1-4	
509	IDLE	IDLE	IDLE	IDLE	UD1-5	
510	IDLE	IDLE	NFC2-H	IDLE	NFC2-H	NFC has priority over user data
511	UD2-1	IDLE	IDLE	IDLE	UD1-6	
512	UD2-2	UFC2-H	IDLE	IDLE	UFC2-H	UFC has priority over user data
513	UD2-3	UFC2-D	IDLE	IDLE	UFC2-D	
514	UD2-4	UFC2-D	IDLE	IDLE	UFC2-D	
515	UD2-5	IDLE	IDLE	IDLE	UD1-7	
516	UD2-6	IDLE	IDLE	IDLE	UD1-8	
517	UD2-7	IDLE	IDLE	IDLE	UD2-1	
518	UD2-8	IDLE	IDLE	IDLE	UD2-2	
519	IDLE	IDLE	NFC3-H	USER K 1	NFC3-H	NFC has priority over USER K
520	IDLE	IDLE			USER K 1	USER K has priority over user data
521		IDLE			UD2-3	
522		IDLE			UD2-4	
523		IDLE			UD2-5	
524					UD2-6	
525					UD2-7	
526					UD2-8	
527					IDLE	

XILINX®

Chapter 4

Language Interface and Test Bench Integration

Aurora 64B/66B BFM (ABFM 64B/66B) is a C++ bus function model for the Aurora 64B/66B protocol that is compiled and distributed as a shared library. The shared library is dynamically linked and simulated with the third-party simulators supporting various hardware language interfaces such as programming language interface (PLI), foreign language interface (FLI), and VHDL procedural interface (VHPI). The language interface provides a standard mechanism to communicate with the simulator through C procedural interface thereby allowing a HDL test bench to invoke modules implemented in C. The language interface defines a large collection of predefined function prototypes and the simulator vendor implements these functions as a part of simulator package. These functions provide a mechanism to access the vendor dependent simulator data corresponding to the HDL objects in the simulation environment in a seamless manner.

The interaction of ABFM 64B/66B with the simulation environment is through a set of user routines in the shared library. The user routines in the shared library make use of the language interface function prototypes to exchange information to/from Aurora architecture model and the HDL test bench. Whenever a particular task implemented as a user routine needs to be carried out, a function or procedural call is made to the corresponding user routine from within the test bench. As the shared libraries are dynamically linked, the simulator should be made aware of the external user routines available in shared libraries at load time so that the simulator can search the shared library whenever a reference to the user routine is made from within the test bench.

ABFM 64B/66B supports the following hardware language interfaces:

- PLI interface for Verilog: Programming language interface (PLI) is an IEEE standard (IEEE 1364 Verilog PLI Standard) which defines a standard and powerful interface to integrate C functions with Verilog. Almost all simulators supporting Verilog language provide PLI library as a part of the simulator package.
- FLI interface: Foreign language interface (FLI) is a proprietary VHDL language interface supported by Mentor Graphics. It provides C procedural access to information within a VHDL simulator. The user routines in shared library make use of the proprietary function prototypes and structures provided by the FLI interface to communicate with the test bench in a VHDL environment. As FLI is Mentor Graphics proprietary, it can be simulated only in ModelSim, the VHDL simulator supported by Mentor Graphics.
- VHPI interface: The VHDL procedural interface (VHPI) is a C procedural interface for VHDL. VHPI is an evolving IEEE standard with a C procedural interface for developing models and applications that are easily integrated into a compliant VHDL simulator. NCSim VHDL simulator from Cadence and VCS VHDL simulator from Synopsis support the VHPI specification standard.

The user routines in the shared library compose the language interface module for ABFM 64B/66B. All three language interfaces share the IDF interface, aurora architecture model and the checker module. The language interface handles only the interface to the simulator exchanging information between the common modules and the simulator. As the language interface is dependent on the language and the simulator, a separate shared library needs to be built from the common modules and the language interface corresponding to the particular simulator and language. Also, as ABFM 64B/66B is distributed as a shared library, it is dependent on the platform and operating system on which the simulation is run. Therefore, ABFM 64B/66B package includes one shared library for each operating system - platform - simulator - language combination supported by the release. For details regarding the platform, operating system, simulator and language supported by the release, refer the release notes, RELNOTES.txt, in the top-level directory of ABFM 64B/66B distribution.

In order to run simulation that interfaces with the ABFM 64B/66B, following components are required:

- The ABFM 64B/66B shared library corresponding to the language, simulator, operating system, and the platform on which the simulation is to run
- The HDL language interface wrapper file which initializes the ABFM 64B/66B at simulation time zero by calling the initialization user routine provided in the shared library. The HDL language interface wrapper file is also responsible for making calls to various other user routines in the shared library at appropriate simulation time or event. Though the user routines can be directly invoked from the HDL test bench and the HDL language interface wrapper file is not strictly needed, it is recommended to instantiate the user routines in a separate HDL wrapper file whose input/output port matches that of the ABFM 64B/66B. From the point of view of HDL test bench, the HDL language interface wrapper file acts as the ABFM 64B/66B module.
- HDL test bench file that defines the simulation environment. Typically, the test bench file generates the resets and clocks, and instantiates the required number of ABFM 64B/66B module as defined by the HDL language interface wrapper file.
- An IDF file for each instance of ABFM 64B/66B module that describes the type and the sequence of packets to be transmitted by the corresponding ABFM 64B/66B instance.

In sections that follow, user routines available in the shared library and their integration into the test bench through the wrapper file are discussed in detail for each language interface. Each language interface has specific mechanism of how the simulator is made aware of the availability of user routines in the shared library and those details are presented as well.

Verilog PLI Interface

PLI user routines in ABFM 64B/66B shared library are invoked from the Verilog test bench through system calls. All system calls in Verilog start with the $ symbol. For example, $display, $monitor and $finish are some of the common system calls built-in to most Verilog simulators. ABFM 64B/66B system calls are prefixed with $bfm_ to differentiate them from the built-in ones. The user makes use of these system calls in the test bench to communicate with the ABFM 64B/66B. A mapping structure in ABFM 64B/66B establishes the correspondence between a system call and its equivalent user routine.

System Calls and Verilog PLI Wrapper

Currently there are two system calls corresponding to the PLI user routines in the shared library - $bfm_initialize and $bfm_execute. Both system calls support multiple instantiations of the ABFM 64B/66B. The very first argument to both system calls is the instance number. For example, to initialize two separate ABFM 64B/66Bs, the following calls can be made:

- First ABFM 64B/66B: $bfm_initialize (0, ….) & $bfm_execute(0)
- Second ABFM 64B/66B: $bfm_initialize (1, ….) & $bfm_execute(1)

$bfm_initialize

Arguments: INSTANCE_NUMBER, NO_OF_LANES, BFM_MODE, INTERFACE_MODE, NFC_MODE, LATENCY, SERIAL_CLK_PERIOD, SERIAL_CLK, TX_BITS, RX_BITS, IDF_PKT_STATUS, APE_STATE, BLOCK_LOCK_ALL_LANES, TX_DATA_OUT_BFM, RX_DATA_IN_BFM

This system call initializes the ABFM 64B/66B and therefore should be called at simulation time zero. When this system call is invoked, the user routine corresponding to the system call probes the simulator to get handles for the simulation objects corresponding to the arguments passed to this system call. Getting handles for simulation objects that need to be communicated through the PLI interface is mandatory. Therefore, this system call expects all parameters, ports, and signals defined in the Verilog PLI wrapper that need to be accessed by ABFM 64B/66B for information exchange be passed as arguments to this system call.

For a given shared library, the parameters, ports, and signals that would be accessed by ABFM 64B/66B to communicate with the simulation environment are fixed, and hence the arguments to this system call are also fixed. Details of arguments of $bfm_initialize system call are listed in Table 4-1.

Table 4-1: **Arguments of $bfm_initialize System Call**

Arguments	Type	Valid Values	Default Value	Description
INSTANCE_NUMBER	Parameter - Integer	0 or 1	0	Instance number of the ABFM 64B/66B
NO_OF_LANES	Parameter - Integer	Any integer between 1 to 24	4	Number of lanes
BFM_MODE	Parameter - String	DUPLEX or SIMPLEX_BOTH	DUPLEX	Mode of operation of ABFM 64B/66B
INTERFACE_MODE	Parameter - String	FRAME or STREAM	FRAME	User interface mode

Table 4-1: **Arguments of $bfm_initialize System Call *(Cont'd)***

Arguments	Type	Valid Values	Default Value	Description
NFC_MODE	Parameter - String	IMMEDIATE or COMPLETION	IMMEDIATE	Type of native flow control
LATENCY	Parameter - Integer	Any integer greater than 0	20	Round trip latency for native flow control
SERIAL_CLK_PERIOD	Parameter - Integer	Any integer greater than 0	320	Clock period of serial clock input (in picoseconds)
SERIAL_CLK	Input Port (1)	Not Applicable	Not Applicable	Serial clock input port
TX_BITS	Signal	Not Applicable	Not Applicable	Same as serial link TX_P output port1. The width of the signal should be same as parameter NO_OF_LANES.
RX_BITS	Signal	Not Applicable	Not Applicable	Same as serial link RX_P input port1. The width of the signal should be same as parameter NO_OF_LANES.
IDF_PKT_STATUS	Output Port (1)	Not Applicable	Not Applicable	Active high signal indicating that the current instance of ABFM 64B/66B has completed transmission of packets. This signal may be used by the simulation environment to terminate the simulation
APE_STATE	Signal	Not Applicable	Not Applicable	2-bit signal indicating the current state of the ABFM 64B/66B initialization state machine. The APE_STATE signal encoding is as follows: 0 - Lane Init state 1 - Channel Bonding state 2 - Wait For Remote state 3 - Channel Ready state
BLOCK_LOCK_ALL_LANES	Signal	Not Applicable	Not Applicable	Signal indicating that all lanes of current instance of ABFM 64B/66B have achieved block lock.
TX_DATA_OUT_BFM	Signal	Not Applicable	Not Applicable	Transmit data of current instance of ABFM 64B/66B before scrambling. The width of this signal should be the value of the parameter NO_OF_LANES times 64.

Table 4-1: **Arguments of $bfm_initialize System Call *(Cont'd)***

Arguments	Type	Valid Values	Default Value	Description
RX_DATA_IN_BFM	Signal	Not Applicable	Not Applicable	Receive data of current instance of ABFM 64B/66B after de-scrambling. The width of this signal should be the value of the parameter NO_OF_LANES times 64.

1. The direction of ports are defined with respect to the Verilog PLI wrapper

The correspondence between the objects (parameter, port, or signal) in Verilog PLI wrapper and the underlying user routine of the system call is through the order in which they appear in the argument list of the $bfm_initialize system call. Therefore, it is important to preserve the order and the semantics of the formal arguments passed to this system call in order to maintain a map between the simulation objects and the internal structures of ABFM 64B/66B. The name of the actual argument passed may differ depending on how they are defined in the Verilog PLI wrapper.

The Verilog PLI wrapper, abfm_pli_wrapper.v, in the ABFM 64B/66B distribution has the following declaration:

```
module ABFM_WRAPPER #(
    parameter INSTANCE_NUMBER = 0,
    parameter NO_OF_LANES = 4,
    parameter BFM_MODE = "DUPLEX",
    parameter INTERFACE_MODE  = "FRAME",
    parameter NFC_MODE = "IMMEDIATE",
    parameter LATENCY = 20,
    parameter SERIAL_CLK_PERIOD = 320
  )
  (
    input SERIAL_CLK,
    input PARALLEL_CLK,
    input [NO_OF_LANES-1 : 0] RX_P,
    input [NO_OF_LANES-1 : 0] RX_N,
    output reg [NO_OF_LANES-1 : 0] TX_P,
    output reg [NO_OF_LANES-1 : 0] TX_N,
    output reg IDF_PKT_STATUS
  );

  reg [NO_OF_LANES-1 : 0]     TX_BITS;
 reg [NO_OF_LANES-1 : 0]     RX_BITS;
 reg [1:0]                   APE_STATE;
 reg                         BLOCK_LOCK_ALL_LANES;
 reg [NO_OF_LANES*64-1 : 0] TX_DATA_OUT_BFM;
 reg [NO_OF_LANES*64-1 : 0] RX_DATA_IN_BFM;
```

The system call, $bfm_initialize is invoked from Verilog PLI wrapper as follows:

```
initial
  begin
    $bfm_initialize
    (
      // Order of the following arguments should not be changed
      INSTANCE_NUMBER,
      NO_OF_LANES,
      BFM_MODE,
```

```
        INTERFACE_MODE,
        NFC_MODE,
        LATENCY,
        SERIAL_CLK_PERIOD,
        SERIAL_CLK,
        TX_BITS,
        RX_BITS,
        IDF_PKT_STATUS,
        APE_STATE,
        BLOCK_LOCK_ALL_LANES,
        TX_DATA_OUT_BFM,
        RX_DATA_IN_BFM
      );
    end
```

As the system call is invoked within the initial block, it is executed only once at the start of the simulation.

$bfm_execute

Arguments: INSTANCE_NUMBER

This system call is easier to understand and use. This system call takes a single argument which is the instance number of ABFM 64B/66B. All handles to the simulation objects accessed by this system call are obtained by the $bfm_initialize call at the start of the simulation. The bulk of the ABFM 64B/66B functionality is exercised by invoking this system call. Each invocation of this system call does the following tasks in order:

1. Fetch all input values from receive serial link
2. Execute RX path
3. Execute TX path
4. Drive all output values to transmit serial link

This system call is invoked from the Verilog PLI wrapper as follows:

```
always @(posedge PARALLEL_CLK)
begin
  $bfm_execute(INSTANCE_NUMBER);
end
```

The above usage calls the $bfm_execute on every clock cycle of PARALLEL_CLK. On every clock cycle, the state machine in the ABFM 64B/66B transition to the next state and updates all the internal signals.

Note: The clock used in the construct above is the parallel clock for the RX path. This is used as the primary clock for the ABFM 64B/66B.

ABFM 64B/66B Integration with Verilog Simulation Environment

In order to run simulation with ABFM 64B/66B, the simulator should be made aware of the availability of shared library. The simulator, then, links to the shared library dynamically at simulation load time so that when a system call is encountered in the test bench the simulator can search the shared library. The exact mechanism of how the simulator is made aware of the existence of shared library varies with the simulator.

ModelSim Simulator

For ModelSim simulator, the shared library for PLI applications can be specified in one of the following ways:

- As an entry in Veriuser option in ModelSim user initialization file
- By setting the environment variable PLIOBJS
- By passing an extra argument '-pli' to the simulator

ABFM 64B/66B simulation infrastructure makes use of the last option of passing an extra argument to make ModelSim aware of the ABFM 64B/66B shared library. The simulation infrastructure generates a ModelSim TCL script for use with the ModelSim simulator. In the ModelSim TCL script, the simulator command, 'vsim', is appended with the '-pli <shared library>'.

A usage example is shown below:

```
vsim -novopt -GSERIAL_CLK_PERIOD=320 \
             -GINTERFACE_MODE="STREAM" \
             -GBFM_MODE="DUPLEX" \
             -GNO_OF_LANES=8 \
             -GNFC_MODE="NONE" ABFM_TB -pli abfm_mti_verilog.so
```

NCSim (NC-Verilog) Simulator

Integrating ABFM 64B/66B dynamically with the NCSim simulator requires that an extra command line option '-loadpli1' be passed to the NCSim elaborator, 'ncelab'. The command line option '-loadpli1' instructs the elaborator to load the shared library and to register the system calls while elaborating. The value passed to the argument '-loadpli1' consists of following two parts:

- The name of the shared library that contains the PLI user routines
- The name of the bootstrap function that registers the system task structure. The system task structure associates the system calls with the corresponding PLI user routines and should be defined in the shared library. The simulator executes the bootstrap function during elaboration phase, that is, just before the simulation starts. In ABFM 64B/66B shared library, the bootstrap function is named as 'abfm_boot_pli'

To load the PLI application, the command line option '-loadpli1' is passed the name of the shared library and the name of the bootstrap function as follows:

```
ncelab -loadpli1 shared_lib:bootstrap_fn
```

The simulator loads the library shared_lib and executes the bootstrap function bootstrap_fn defined in the shared_lib.

ABFM 64B/66B simulation infrastructure generates a shell script for use with the NCSim NC-Verilog simulator. In the shell script, the elaborator command, 'ncelab', is passed the required '-loadpli1 shared library:bootstrap_fn'.

A usage example is shown below:

```
ncelab -RELAX -ACCESS +RWC \
       -defparam ABFM_TB.SERIAL_CLK_PERIOD=320 \
       -defparam ABFM_TB.INTERFACE_MODE="STREAM" \
       -defparam ABFM_TB.BFM_MODE="DUPLEX" \
       -defparam ABFM_TB.NO_OF_LANES=8 \
       -defparam ABFM_TB.NFC_MODE="NONE" \
       -LOADPLI1 abfm_ncsim_verilog.so:abfm_boot_pli \
       work.ABFM_TB
```

VHDL FLI/VHPI Interface

FLI or VHPI user routines in ABFM 64B/66B shared library are invoked from the VHDL test bench through VHDL procedure calls. The mapping between the FLI/VHPI user routines in ABFM 64B/66B and the corresponding procedure call used in the VHDL test bench is established by a foreign subprogram. A foreign subprogram is a VHDL function or procedure call declaration with the attribute FOREIGN attached to it. Foreign subprograms do not need to have VHDL function bodies or procedure definitions. The function or procedure definition for foreign subprogram is written in C. If a VHDL definition does exist for the foreign subprogram declaration, then the simulator may choose to ignore it. The attribute FOREIGN attached to the declaration indicates to the simulator that the definition for the declaration is in C. The foreign subprogram must have equivalent VHDL parameters as arguments that match with the user routine in order as well as in associated type.

In the VHDL test bench, the user makes use of these foreign subprogram calls to communicate with the ABFM 64B/66B. Each user routine should have a corresponding foreign subprogram declared for it to be called from within the test bench. In ABFM 64B/66B distribution, the foreign subprograms for the user routines are declared in the FLI or VHPI wrappers.

The user routines in ABFM 64B/66B and the corresponding foreign subprograms are prefixed with bfm_ to differentiate them from the built-in or any third party supplied procedures that might be in use in the simulation environment. In Verilog PLI interface, the user routines in ABFM 64B/66B is interfaced to the test bench through system calls and they start with the $ symbol. However, in FLI/VHPI interface, there is no such requirement to start the foreign subprograms with any special characters.

Though the VHDL language interface is different for different simulators (FLI for ModelSim and VHPI for NCSim) the application programmers interface in terms of available user routines, their signatures and hence, foreign subprogram declarations remains the same for both interfaces. Therefore, in section that follows, foreign subprogram declarations for both interfaces and their integration with the FLI/VHPI wrappers are discussed in common.

Foreign Subprograms and VHDL FLI/VHPI Wrapper

Currently there are three foreign subprograms corresponding to the FLI/VHPI user routines in the shared library: bfm_initialize, bfm_serial_bits and bfm_execute. All three system calls support multiple instantiations of the ABFM 64B/66B. The very first argument to all three system calls is the instance number.

bfm_initialize

Arguments: INSTANCE_NUMBER, NO_OF_LANES, BFM_MODE, INTERFACE_MODE, NFC_MODE, LATENCY, SERIAL_CLK_PERIOD

This foreign subprogram initializes the ABFM 64B/66B and therefore should be called at simulation time zero. All arguments passed to this foreign subprogram are VHDL generics. When this foreign subprogram is invoked, the user routine corresponding to the foreign subprogram probes the simulator to get values for the generics corresponding to the arguments passed to this foreign subprogram and stores them internally for use by ABFM 64B/66B.

For a given shared library, the generics that would be accessed by ABFM 64B/66B to communicate with the simulation environment are fixed and hence the arguments to this

foreign subprogram are also fixed. Details of arguments of bfm_initialize foreign subprogram are listed in Table 4-2, page 55.

Table 4-2: **Arguments of Foreign Subprogram bfm_initialize**

Arguments	Type	Mode	Valid Values	Default Value	Description
INSTANCE_NUMBER	Generic (1) - Integer	Input	0 or 1	0	Instance number of the ABFM 64B/66B
NO_OF_LANES	Generic (1) - Integer	Input	Any integer between 1 to 24	4	Number of lanes
BFM_MODE	Generic (1) - String	Input	DUPLEX or SIMPLEX_BOTH	DUPLEX	Mode of operation of ABFM 64B/66B
INTERFACE_MODE	Generic (1) - String	Input	FRAME or STREAM	FRAME	User interface mode
NFC_MODE	Generic (1) - String	Input	IMMEDIATE or COMPLETION	IMMEDIATE	Type of native flow control
LATENCY	Generic (1) - Integer	Input	Any integer greater than 0	20	Round trip latency for native flow control
SERIAL_CLK_PERIOD	Generic (1) - Integer	Input	Any integer greater than 0	320	Clock period of serial clock input (in picoseconds)

1. The type declaration of formal arguments of bfm_initialize foreign subprogram is variable. However, the actual arguments passed are VHDL generics assigned to temporary variables.

The correspondence between the simulation object in VHDL FLI/VHPI foreign subprogram arguments and the arguments of underlying user routine is through the order in which they appear in the argument list of the foreign subprogram. Therefore, it is important to preserve the order and the semantics of the formal arguments passed to this foreign subprogram in order to maintain a map between the simulation objects and the internal structures of ABFM 64B/66B. The name of the actual argument passed may differ depending on how the corresponding generics are defined in the VHDL FLI/VHPI wrapper.

The VHDL FLI wrapper, abfm_fli_lin_wrapper.vhd in the ABFM 64B/66B distribution should be used when simulating with ModelSim simulator in VHDL simulation environment on Linux platforms. It has the following foreign subprogram declaration for bfm_initialize user routine:

```
procedure bfm_initialize (
    variable INSTANCE_NUMBER   : IN integer;
    variable NO_OF_LANES       : IN integer;
    variable BFM_MODE          : IN string;
    variable INTERFACE_MODE    : IN string;
    variable NFC_MODE          : IN string;
    variable LATENCY           : IN integer;
    variable SERIAL_CLK_PERIOD : IN integer
  ) is
  begin
    assert false report \
       "ERROR: foreign subprogram bfm_intialize not called" severity
note;
  end;
  attribute FOREIGN of bfm_initialize: procedure is "bfm_initialize
abfm_mti_vhdl.so";
```

The attribute FOREIGN is of type string and the value of the string is simulator dependent. For ModelSim, the value of the string contains two parts separated by a space; First part of the string specifies the user routine associated with the function or the procedure and the second part specifies the name of the shared library that contains the user routine.

There are two separate FLI wrappers; one to be used when running simulation on Linux and the other when running simulation on Windows. The two wrappers are identical but for the FOREIGN attribute specification. As the extension of the shared library differs on Linux (.so) and Windows (.dll), the wrapper needs to specify the extension corresponding to the operating system on which the simulation is run. In the FLI wrapper for Windows, abfm_fli_nt_wrapper.vhd in ABFM 64B/66B distribution, the FOREIGN attribute specification in the above code snippet reads as follows:

```
attribute FOREIGN of bfm_initialize: procedure is "bfm_initialize
abfm_mti_vhdl.dll";
```

The VHDL VHPI wrapper, abfm_vhpi_wrapper.vhd in the ABFM 64B/66B distribution should be used when simulating with NCSim simulator in VHDL simulation environment on Linux platforms. The VHPI wrapper has the following foreign subprogram declaration for bfm_initialize user routine:

```
procedure bfm_initialize (
    variable INSTANCE_NUMBER   : IN integer;
    variable NO_OF_LANES       : IN integer;
    variable BFM_MODE          : IN string;
    variable INTERFACE_MODE    : IN string;
    variable NFC_MODE          : IN string;
    variable LATENCY           : IN integer;
    variable SERIAL_CLK_PERIOD : IN integer
  );
  attribute FOREIGN of bfm_initialize : procedure is "VHPIDIRECT
abfm_ncsim_vhdl.so:  bfm_initialize";
```

NCSim simulator also expects the value of the FOREIGN attribute to be a string. But the string contains three parts instead of two. The first part of the string is the keyword 'VHPIDIRECT' which indicates to the simulator that it should use direct VHDL binding to the user routines in C. The keyword 'VHPIDIRECT' is separated from the rest of the string by a space. The other two parts of the string specifies the name of the shared library and the user routine associated with the procedure (or function) declaration in that order. A colon separates the last two parts of the string.

The FLI wrapper (abfm_fli_lin_wrapper.vhd or abfm_fli_nt_wrapper.vhd) and the VHPI wrapper (abfm_vhpi_wrapper.vhd) in the ABFM 64B/66B distribution has the following entity declaration:

```
entity  ABFM_WRAPPER is
  generic (
    INSTANCE_NUMBER   : integer := 0;
    NO_OF_LANES       : integer := 1;
    BFM_MODE          : string  := "DUPLEX";
    INTERFACE_MODE    : string  := "FRAME";
    NFC_MODE          : string  := "IMMEDIATE";
    LATENCY           : integer := 20;
    SERIAL_CLK_PERIOD : integer := 320
  );
  port (
    SERIAL_CLK        : IN  std_logic;
    PARALLEL_CLK      : IN  std_logic;
    TX_P              : OUT std_logic_vector(NO_OF_LANES-1 downto 0);
   TX_N           : OUT std_logic_vector(NO_OF_LANES-1 downto 0);
```

```
    RX_P           : IN  std_logic_vector(NO_OF_LANES-1 downto 0);
    RX_N           : IN  std_logic_vector(NO_OF_LANES-1 downto 0);
     IDF_PKT_STATUS    : OUT std_logic
  );
end ABFM_WRAPPER;
```

The foreign subprogram, bfm_initialize is invoked from FLI/VHPI wrapper as follows:

```
P1: process

    variable v_INSTANCE_NUMBER   : integer;
    variable v_NO_OF_LANES       : integer;
    variable v_BFM_MODE          : string(1 to BFM_MODE'length);
    variable v_INTERFACE_MODE    : string(1 to INTERFACE_MODE'length);
    variable v_NFC_MODE          : string(1 to NFC_MODE'length);
    variable v_LATENCY           : integer;
    variable v_SERIAL_CLK_PERIOD : integer;

  begin

    v_INSTANCE_NUMBER   := INSTANCE_NUMBER;
    v_NO_OF_LANES       := NO_OF_LANES;
    v_BFM_MODE          := BFM_MODE;
    v_INTERFACE_MODE    := INTERFACE_MODE;
    v_NFC_MODE          := NFC_MODE;
    v_LATENCY           := LATENCY;
    v_SERIAL_CLK_PERIOD := SERIAL_CLK_PERIOD;

    bfm_initialize(
      v_INSTANCE_NUMBER,
      v_NO_OF_LANES,
      v_BFM_MODE,
      v_INTERFACE_MODE,
      v_NFC_MODE,
      v_LATENCY,
      v_SERIAL_CLK_PERIOD
    );

    wait;
  end process P1;
```

The formal arguments of bfm_initialize are declared as variables in the foreign subprogram declaration. Hence, the actual arguments passed while calling bfm_initialize should also be defined as variable and must match the variable type. Therefore, the generics passed from the test bench to the FLI/VHPI wrappers are assigned to temporary variables and these variables are then passed as actual argument to the bfm_initialize call.

Since the purpose of bfm_initialize is to initialize the ABFM 64B/66B, it is expected to be called only once at the start of the simulation. The 'wait' statement at the end of the process ensures that the bfm_initialize is executed only once at the start of the simulation.

bfm_serial_bits

Arguments: INSTANCE_NUMBER, RX_BITS, TX_BITS

This system call is used to exchange data with the serial links - to drive transmit serial bits or to get receive serial bits. Therefore, this procedure is typically called on every clock cycle of serial clock. PLI interface provides a mechanism to define a hook to an internal function in the shared library that should be called whenever a simulation event occurs. In the

ABFM 64B/66B, the internal function hooked to the simulation event is responsible for receiving serial bits on every serial clock in a manner transparent to the user. However, FLI or VHPI interface do not provide any such mechanism to hook a function to the simulation event. Therefore, the user should explicitly call bfm_serial_bits in the simulation environment on every serial clock to send or receive data.

Details of arguments of bfm_serial_bits foreign subprogram are listed in Table 4-3.

Table 4-3: **Arguments of Foreign Subprogram bfm_serial_bits**

Arguments	Type	Mode	Valid Values	Default Value	Description
INSTANCE_NUMBER	Generic (1) - Integer	Input	0 or 1	0	Instance number of the ABFM 64B/66B
RX_BITS	Signal (1) - Standard Logic Vector	Input	Not Applicable	Not Applicable	Same as serial link RX_P input port. The width of the signal should be same as parameter NO_OF_LANES.
TX_BITS	Signal (1) - Standard Logic Vector	Output	Not Applicable	Not Applicable	Same as serial link TX_P output port. The width of the signal should be same as parameter NO_OF_LANES.

1. The type declaration of formal arguments of bfm_serial_bits foreign subprogram is variable. However, the actual arguments passed are either VHDL generics or signals assigned to temporary variables.

The VHDL FLI wrapper, abfm_fli_lin_wrapper.vhd in the ABFM 64B/66B distribution has the following foreign subprogram declaration for bfm_serial_bits user routine:

```
procedure bfm_serial_bits (
   variable INSTANCE_NUMBER : IN  integer;
   variable RX_BITS : IN  std_logic_vector(NO_OF_LANES-1 downto 0);
   variable TX_BITS : OUT std_logic_vector(NO_OF_LANES-1 downto 0)
  ) is
  begin
    assert false report "ERROR: foreign subprogram bfm_serial_bits not
called" severity note;
  end;
  attribute FOREIGN of bfm_serial_bits : procedure is "bfm_serial_bits
abfm_mti_vhdl.so";
```

In the FLI wrapper for Windows, abfm_fli_nt_wrapper.vhd in ABFM 64B/66B distribution, the foreign subprogram declaration for bfm_serial_bits is identical to the above code snippet but for the FOREIGN attribute specification. The FOREIGN attribute specification in FLI wrapper for windows reads as follows:

```
attribute FOREIGN of bfm_serial_bits : procedure is "bfm_serial_bits
abfm_mti_vhdl.dll";
```

The VHDL VHPI wrapper, abfm_vhpi_wrapper.vhd in the ABFM 64B/66B distribution has the following foreign subprogram declaration for bfm_serial_bits user routine:

```
procedure bfm_serial_bits (
    variable INSTANCE_NUMBER: IN  integer;
    variable RX_BITS : IN  std_logic_vector(NO_OF_LANES-1 downto 0);
    variable TX_BITS : OUT std_logic_vector(NO_OF_LANES-1 downto 0)
  );
  attribute FOREIGN of bfm_serial_bits : procedure is "VHPIDIRECT
abfm_ncsim_vhdl.so: bfm_serial_bits";
```

The foreign subprogram, bfm_serial_bits is invoked from FLI/VHPI wrapper as follows:

```
P2: process (SERIAL_CLK, RX_BITS)

    variable v_INSTANCE_NUMBER: integer;
    variable v_RX_BITS : std_logic_vector(NO_OF_LANES-1 downto 0);
    variable v_TX_BITS : std_logic_vector(NO_OF_LANES-1 downto 0);

  begin

    v_INSTANCE_NUMBER := INSTANCE_NUMBER;
    v_RX_BITS         := RX_BITS;

    if SERIAL_CLK'EVENT and SERIAL_CLK = '1' then
      bfm_serial_bits(
        v_INSTANCE_NUMBER,
        v_RX_BITS,
        v_TX_BITS
      );
    end if;

    TX_BITS <= v_TX_BITS;

  end process P2;
```

The above usage calls the bfm_serial_bits foreign subprogram on every clock cycle of SERIAL_CLK. On every clock cycle, the data on receive serial links are stored in internal structures of ABFM 64B/66B. Similarly, the data to be transmitted are driven onto transmit serial links.

Note: The clock used in the construct above is the serial clock for the TX/RX path. This clock is used primarily only to transmit and receive serial data.

bfm_execute

Arguments: INSTANCE_NUMBER, IDF_PKT_STATUS, APE_STATE, BLOCK_LOCK_ALL_LANES, TX_DATA_OUT_BFM, RX_DATA_OUT_BFM

This system call is easier to understand and use. The bulk of the ABFM 64B/66B functionality is exercised by invoking this system call. Each invocation of this system call does the following tasks in order:

1. Re-order receive data from serial link into parallel data
2. Execute RX path
3. Execute TX path
4. Order parallel data into serial stream to be driven onto transmit serial link
5. Update values of the internal signal to be displayed onto simulator waveform

Details of arguments of bfm_execute foreign subprogram are listed in Table 4-4.

Table 4-4: **Arguments of Foreign Subprogram bfm_execute**

Arguments	Type	Mode	Valid Values	Default Value	Description
INSTANCE_NUMBER	Generic (1) - Integer	Input	0 or 1	0	Instance number of the ABFM 64B/66B
IDF_PKT_STATUS	Signal (1) - Standard Logic Vector (2)	Output	Not Applicable	Not Applicable	Active high signal indicating that the current instance of ABFM 64B/66B has completed transmission of packets. This signal may be used by the simulation environment to terminate the simulation
APE_STATE	Signal (1) - Standard Logic Vector (2	Output	Not Applicable	Not Applicable	2-bit signal indicating the current state of the ABFM 64B/66B initialization state machine. The APE_STATE signal encoding is as follows: 0 - Lane Init state 1 - Channel Bonding state 2 - Wait For Remote state 3 - Channel Ready state
BLOCK_LOCK_ALL_LANES	Signal (1) - Standard Logic Vector (2	Output	Not Applicable	Not Applicable	Signal indicating that all lanes of current instance of ABFM 64B/66B have achieved block lock.
TX_DATA_OUT_BFM	Signal (1) - Standard Logic Vector (2	Output	Not Applicable	Not Applicable	Transmit data of current instance of ABFM 64B/66B before scrambling. The width of this signal should be the value of the parameter NO_OF_LANES times 64.
RX_DATA_IN_BFM	Signal (1) - Standard Logic Vector (2	Output	Not Applicable	Not Applicable	Receive data of current instance of ABFM 64B/66B after de-scrambling. The width of this signal should be the value of the parameter NO_OF_LANES times 64.

1. The type declaration of formal arguments of bfm_execute foreign subprogram is variable. However, the actual arguments passed are either VHDL generics or signals assigned to temporary variables.
2. All outputs are declared as standard logic vector. Even single bit signals are defined as vector - standard logic vector(0 downto 0)

The VHDL FLI wrapper, abfm_fli_lin_wrapper.vhd in the ABFM 64B/66B distribution has the following foreign subprogram declaration for bfm_execute user routine:

```
procedure bfm_execute (
    variable INSTANCE_NUMBER      : IN  integer;
    variable IDF_PKT_STATUS       : OUT std_logic_vector(0 downto 0);
    variable APE_STATE            : OUT std_logic_vector(1 downto 0);
    variable BLOCK_LOCK_ALL_LANES : OUT std_logic_vector(0 downto 0);
   variable TX_DATA_OUT_BFM     : OUT std_logic_vector(NO_OF_LANES*64-
1 downto 0);
    variable RX_DATA_IN_BFM      : OUT std_logic_vector(NO_OF_LANES*64-
1 downto 0)
  ) is
  begin
    assert false report "ERROR: foreign subprogram bfm_execute not
called" severity note;
  end;
  attribute FOREIGN of bfm_execute : procedure is "bfm_execute
abfm_mti_vhdl.so";
```

In the FLI wrapper for Windows, abfm_fli_nt_wrapper.vhd in ABFM 64B/66B distribution, the foreign subprogram declaration for bfm_execute is identical to the above code snippet but for the FOREIGN attribute specification. The FOREIGN attribute specification in FLI wrapper for windows reads as follows:

```
attribute FOREIGN of bfm_execute : procedure is "bfm_execute
abfm_mti_vhdl.dll";
```

The VHDL VHPI wrapper, abfm_vhpi_wrapper.vhd in the ABFM 64B/66B distribution has the following foreign subprogram declaration for bfm_execute user routine:

```
procedure bfm_execute(
    variable INSTANCE_NUMBER      : IN  integer;
    variable IDF_PKT_STATUS       : OUT std_logic_vector(0 downto 0);
    variable APE_STATE            : OUT std_logic_vector(1 downto 0);
    variable BLOCK_LOCK_ALL_LANES : OUT std_logic_vector(0 downto 0);
   variable TX_DATA_OUT_BFM     : OUT std_logic_vector(NO_OF_LANES*64-
1 downto 0);
    variable RX_DATA_IN_BFM      : OUT std_logic_vector(NO_OF_LANES*64-
1 downto 0)
  );
  attribute FOREIGN of bfm_execute : procedure is "VHPIDIRECT
abfm_ncsim_vhdl.so: bfm_execute";
```

The foreign subprogram, bfm_execute is invoked from FLI/VHPI wrapper as follows:

```
P3: process (PARALLEL_CLK)

    variable v_INSTANCE_NUMBER      : integer;
    variable v_IDF_PKT_STATUS       : std_logic_vector(0 downto 0);
    variable v_APE_STATE            : std_logic_vector(1 downto 0);
    variable v_BLOCK_LOCK_ALL_LANES : std_logic_vector(0 downto 0);
    variable v_TX_DATA_OUT_BFM      : std_logic_vector(NO_OF_LANES*64-
1 downto 0);
    variable v_RX_DATA_IN_BFM       : std_logic_vector(NO_OF_LANES*64-
1 downto 0);

  begin

    v_INSTANCE_NUMBER := INSTANCE_NUMBER;
```

```
    if PARALLEL_CLK'EVENT and PARALLEL_CLK = '1' then
      bfm_execute(
        v_INSTANCE_NUMBER,
        v_IDF_PKT_STATUS,
        v_APE_STATE,
        v_BLOCK_LOCK_ALL_LANES,
        v_TX_DATA_OUT_BFM,
        v_RX_DATA_IN_BFM
      );
    end if;

    IDF_PKT_STATUS         <=  v_IDF_PKT_STATUS(0);
    APE_STATE              <=  v_APE_STATE;
    BLOCK_LOCK_ALL_LANES   <=  v_BLOCK_LOCK_ALL_LANES(0);
    TX_DATA_OUT_BFM        <=  v_TX_DATA_OUT_BFM;
    RX_DATA_IN_BFM         <=  v_RX_DATA_IN_BFM;

  end process P3;
```

The above usage calls the bfm_execute foreign subprogram on every clock cycle of PARALLEL_CLK. On every clock cycle, the state machine in the ABFM 64B/66B transition to the next state and updates all the internal signals.

Note: The clock used in the construct above is the parallel clock for the RX path. This is used as the primary clock for the ABFM 64B/66B.

ABFM 64B/66B Integration with VHDL Simulation Environment

As the ABFM 64B/66B shared library is linked dynamically at simulation load time, the simulator must be made aware of the availability of the shared library. This allows the simulator to search the shared library for the user routine when the associated procedure call is encountered in the test bench. As with Verilog PLI interface, the exact mechanism of how the simulator is made aware of the existence of shared library varies with the simulator.

ModelSim Simulator

For ModelSim simulator, there are no specific requirements to make the simulator aware of the shared library. As the name of the shared library is specified in the FOREIGN attribute of the foreign subprogram declaration, the simulator is automatically made aware of the shared library in which the user routine is defined. ModelSim simulator searches for the shared library in the following locations in the order given:

- Current directory in which the simulation is run
- In the location specified in FOREIGN attribute if full path to the shared library is given
- In Linux, in the directories specified using $LD_LIBRARY_PATH environment variable

NCSim (NC-VHDL) Simulator

For NCSim simulator, the shared library for VHPI applications can be specified in one of the following ways:

- By passing an extra argument '-loadvhpi' to the NCSim elaborator, 'ncelab'. The value passed to the argument '-loadvhpi' consists of following two parts:
  - The name of the shared library that contains the VHPI user routines

- The name of the bootstrap function. The simulator executes the bootstrap function during elaboration phase, that is, just before the simulation starts. Only the bootstrap function is executed during elaboration phase; the other foreign subprograms are executed during simulation time. In ABFM 64B/66B shared library, the bootstrap function is named as 'abfm_boot_vhpi' and is a dummy function that does nothing.

  To load the VHPI application, the command line option '-loadvhpi' is passed the name of the shared library and the name of the bootstrap function as follows:

  ```
  ncelab -loadvhpi shared_lib:bootstrap_fn
  ```

  The simulator loads the library shared_lib and executes the bootstrap function bootstrap_fn defined in the shared_lib.

- As an option passed to FOREIGN attribute in foreign subprogram declaration

In either case, the shared library may be specified by its name or by its path. If the path to the shared library is specified, it may be either absolute or relative. If only the name of the shared library is specified, then, the shared library is searched in the directories specified using $LD_LIBRARY_PATH environment variable. If the shared library is already loaded with '-loadvhpi' option, then, it should not be specified again using the FOREIGN attribute.

ABFM 64B/66B simulation infrastructure makes use of the last option of passing the name of the shared library as an option to FOREIGN attribute to make NCSim simulator aware of the existence of ABFM 64B/66B shared library. Though dummy bootstrap function 'abfm_boot_vhpi' is defined in the ABFM 64B/66B it is never loaded and executed with '-loadvhpi' option.

XILINX®

Chapter 5

Simulating with Aurora 64B/66B BFM

The Aurora 64B/66B bus functional model (ABFM 64B/66B) is distributed as a shared library with the required simulation infrastructure to support seamless integration of shared library with the third-party simulators such as ModelSim and NCSim. The simulation infrastructure creates an easy and efficient means to simulate and analyze the results of ABFM 64B/66B simulation with various simulators. ABFM 64B/66B simulation infrastructure consists of the following components:

- The wrapper for various language interface
- Top-level test bench for BFM talk-mode (for BFM-BFM simulation), both in Verilog and VHDL
- Sample testcases for packet generation - A set of IDF files for various test scenarios in terms of packet types, their size, data, and sequence
- A set of Perl and shell (or command for Windows) scripts that sets up the simulation environment and runs the simulation

The simulation infrastructure provided in ABFM 64B/66B distribution enables the users to get the simulations up and running quickly with minimal changes to the simulation environment.

Simulation Procedure

To simulate ABFM 64B/66B in talk-mode (BFM-BFM), follow the step-by-step instructions given below:

1. ABFM 64B/66B is distributed as a zipped tar ball for Linux environment and as a zip file for the Windows environment. Extract the ABFM 64B/66B files from the distribution .zip file.
   - On Linux:

     ```
     bash$ tar -zxvf <distribution file>
     ```
   - On Windows:

     Use a program that can unzip .zip files. Examples include 7-Zip and WinZip.

   The distribution file will be extracted to ABFM 64B/66B installation directory named abfm64b66b-<release version>
2. Set the environment variable BFMDIR to point to the ABFM 64B/66B installation directory - the top level directory where the ABFM 64B/66B zip file is extracted:
   - On Linux
     - Bash Shell

       ```
       bash$ export BFMDIR=<Aurora 64B66B BFM installation directory>
       ```

- Csh Shell

```
csh$ setenv BFMDIR <Aurora 64B66B BFM installation directory>
```

- On Windows

```
C:\BFM INSTAL DIR\BIN> SET BFMDIR=<Aurora 64B66B BFM installation directory>
```

3. Navigate to bin directory in the ABFM 64B/66B installation directory
4. Run the Perl script, run_test.pl. This Perl script manages the bulk of the required setup for a given simulation run, thereby making ABFM 64B/66B relatively easy to use. Following are a list of tasks carried out by this script along with its package file, bfm_bfm_pkg.pm:

- Creates a working space for simulation. In the top-level directory pointed by BFMDIR, a directory named 'run' is created. Under directory 'run', two sub directories - 'sim', for current simulation run and 'results' for saving the simulation results - are created.
- Sets up required environment variables such as LD_LIBRARY_PATH
- Copies the necessary files to simulation directory. Following files are copied to the simulation directory:
  - Top-level test bench
  - Shared library and the wrapper file corresponding to the simulator and the language of users choice and the platform on which the simulation is run
  - IDF file corresponding to the testcases to be run.
- Generates simulator specific scripts in the simulation directory to compile and load the necessary files
- Loads the top-level test bench and ABFM 64B/66B shared library and runs the simulation with the testcases described in the IDF files
- At the end of the simulation, invokes a check routine that analyzes the log file for any possible IDF parsing errors and protocol violations. The check routine also compares the packet data logged by the ABFM 64B/66B for data integrity. Based on the analysis of the simulation run, a report file, report.txt is generated.
- Saves the current simulation directory in the results directory for further analysis. The name of the directory is based on the status of the results of the simulation analysis and will reveal whether the simulation is a pass or fail.

To run the Perl script, run_test.pl, issue the following command at the command prompt:

```
Perl run_test.pl -chl_partner=<chl_partner>
                 -num_lanes=<num_lanes>
                 -bfm_mode=<bfm_mode>
                 -interface_mode=<interface_mode>
                 -nfc_mode=<nfc_mode>
                 -ser_clk_period=<serial_clk_period>
                 -bfm_testcase=<bfm_testcase>
                 -partner_testcase=<partner_testcase>
                 -language=<language>
                 -sim=<simulator>
                 -platform=<platform>
                 -batch_mode
                 -debug=2
```

Arguments of run_test.pl script is described in Table 5-1.

Table 5-1: **Arguments of Script, run_test.pl**

Argument	Valid Value	Description
chl_partner	BFM	Indicates the mode of channel partner.
num_lanes	Any integer between 1 to 24	Indicates the number of lanes for which ABFM 64B/66B is configured for.
bfm_mode	DUPLEX or SIMPLEX_BOTH	Indicates whether the ABFM 64B/66B should be configured as simplex or duplex.
interface_mode	FRAME or STREAM	Indicates whether the ABFM 64B/66B should be configured to support framing interface or streaming interface.
nfc_mode	IMMEDIATE or COMPLETION	Indicates the type of NFC support ABFM 64B/66B is configured for.
ser_clk_period	Any integer	Indicates the serial clock period in picoseconds.
bfm_testcase	Any string (1)	Indicates the testcase that should be applied on transmit side of ABFM 64B/66B instance 0. Only testcase name is required; Script derives the path internally.
partner_testcase	Any string (1)	Indicates testcase that should be applied on transmit side of ABFM 64B/66B instance 1. Only testcase name is required; Script derives the path internally.
language	verilog or vhdl	Indicates the language for the current simulation run.
sim	mti or ncsim	Indicates the simulator for the current simulation run.
platform	For mti, lin or nt For ncsim, lin only	Indicates the platform for the current simulation run.
batch_mode	Not applicable (2)	Indicates the simulator should be invoked in batch mode rather than in GUI mode. Without this option, simulator will be invoked in GUI mode.
debug	1 or 2	Indicates the level of verbosity of messages.

1. The string should refer to the name of the testcase. It may be one of the seven sample testcases provided in ABFM 64B/66B distribution or may be a new testcase developed by the user. If the user develops a new testcase, then it must be placed in the same testcase directory where the sample testcases reside. For each new testcase, a directory with the name of the testcase should be created in the testcase directory and two IDF files should be placed in it with the name idffile0 and idffile1 describing the packet sequence that should be applied to ABFM 64B/66B instance 0 and 1 respectively. This is required as the script internally derives the path for the testcase IDF file(s).
2. This option does not take value.

Running the Perl script as, 'perl run_test.pl -help', at the command prompt displays a usage guide with a brief description of its argument.

Examples:

1. To run BFM-BFM talk mode batch simulation on Verilog-ModelSim-Linux platform with both ABFM 64B/66B configured as 4-lane supporting duplex, framing, and NFC-Immediate interface at a line rate of 3.125 Gbps with ABFM 64B/66B instance 0 injecting traffic as per description of sample testcase 'All_Traffic_Test' and ABFM 64B/66B instance 1 injecting traffic as per description of sample testcase 'UFC_Test', invoke the script run_test.pl as:

```
bash$ perl run_test.pl -chl_partner=BFM -num_lanes=4 -bfm_mode=DUPLEX \
                       -interface_mode=FRAME -nfc_mode=IMMEDIATE \
                       -ser_clk_period=320 \
                       -bfm_testcase=All_Traffic_Test \
                       -partner_testcase=UFC_Test \
                       -language=verilog -sim=mti -platform=linux \
                       -batch_mode -debug=2
```

2. To run BFM-BFM talk mode GUI simulation on VHDL-NCSim-Windows platform with both ABFM 64B/66Bs configured as single lane supporting simplex, streaming, and NFC-Completion interface at a line rate of 4.5 Gbps with ABFM 64B/66B instance 0 injecting traffic as per description of sample testcase 'Embedded_UFC_Test' and ABFM 64B/66B instance 1 injecting traffic as per description of sample testcase 'USERK_Test', invoke the script run_test.pl as:

```
C:\BFMDIR\bin> perl run_test.pl -chl_partner=BFM -num_lanes=1 \
                                -bfm_mode=SIMPLEX_BOTH \
                                -interface_mode=STREAM \
                                -nfc_mode=COMPLETION \
                                -ser_clk_period=222 \
                                -bfm_testcase=Embedded_UFC_Test \
                                -partner_testcase=USERK_Test \
                                -language=vhdl -sim=ncsim -platform=nt \
                                -debug=2
```

In order to further ease the process of getting the ABFM 64B/66B simulations up and running quickly, a wrapper script has been provided in the ABFM 64B/66B distribution that sets the BFMDIR environment variable and runs all the seven sample testcases provided in the distribution on ABFM 64B/66B instance 0 against a fixed testcase running on ABFM 64B/66B instance 1. User may pass the testcase to run on ABFM 64B/66B instance 1 as an argument failing which PDU_Test is chosen as default. The user is required to do minor edits to the script - all variables that would require edits are within double square brackets. By default the language is chosen to be Verilog and the simulator, ModelSim. User may change these default options to reflect their choice of language and simulator.

Depending on whether the distribution is for Linux or Windows platform, the format of the wrapper script will vary. In the distribution for Linux platform, the wrapper script, run_bfm_bfm.sh is a bash shell script. In the distribution for Windows platform, the wrapper script run_bfm_bfm.bat is a Windows command script. The script is invoked as follows (after editing the variables in double square bracket to a valid value):

- On Linux
  - To run all seven sample testcases on ABFM 64B/66B instance 0 and PDU_Test on ABFM 64B/66B instance 1

    ```
    bash$ ./run_bfm_bfm.sh
    ```

  - To run all seven sample testcases on ABFM 64B/66B instance 0 and UFC_Test on ABFM 64B/66B instance 1

    ```
    bash$ ./run_bfm_bfm.sh UFC_Test
    ```

- On Windows Command Prompt
  - To run all seven sample testcases on ABFM 64B/66B instance 0 and PDU_Test on ABFM 64B/66B instance 1

    ```
    C:\BFMDIR\bin> run_bfm_bfm.bat
    ```

  - To run all seven sample testcases on ABFM 64B/66B instance 0 and UFC_Test on ABFM 64B/66B instance 1

    ```
    C:\BFMDIR\bin> run_bfm_bfm.bat UFC_Test
    ```

Analyzing Results

The top-level test bench in ABFM 64B/66B terminates the simulation when all the packet requests in IDF files of both instances of ABFM 64B/66B are serviced. Once the simulation terminates, the script run_test.pl saves the current simulation run in a results directory under '<BFM install directory>/run/results'. The results directory created by the script has the following syntax:

```
<pass_status>_<bfm_testcase>_<num_lanes>_<bfm_mode>_<interface_mode>_<
nfc_mode>_<timestamp>
```

where pass_status is the status of the result of the simulation run and timestamp is the time at which the simulation run completed. The rest of the components in the directory name - bfm_testcase, num_lanes, bfm_mode, interface_mode, and nfc_mode - all take their meaning as explained in Table 5-1, page 67.

Example:

1. The result directory of a successful simulation where the ABFM 64B/66B are configured as 4 lane supporting duplex, streaming, and NFC-Complete interface with ABFM 64B/66B instance 0 injecting traffic as per description of sample testcase 'Embedded_UFC_Test' will be named as

   ```
   PASSED_Embedded_UFC_Test_4_DUPLEX_STREAM_COMPLETION_17Jan_17:17
   ```

   (Assuming the simulation completes on 17th January at 17:17)

2. If the above simulation fails, then the results directory will be named as

   ```
   FAILED_Embedded_UFC_Test_4_DUPLEX_STREAM_COMPLETION_17Jan_17:17
   ```

The files in the results directory are described in Table 5-2. Only the files that are created by ABFM 64B/66B or the files that are created or copied from other locations by the script run_test.pl are listed in the table. Apart from these files, the results directory will also contain simulator specific log files.

Table 5-2: **Files in Results Directory**

Name of the File	Description
bfm0.log	This file is generated by ABFM 64B/66B and contains all output messages from ABFM 64B/66B instance 0. The verbosity of logging can be controlled by setting verbosity parameters in IDF file corresponding to ABFM 64B/66B instance 0 (idffile0)
bfm1.log	This file is generated by ABFM 64B/66B and contains all output messages from ABFM 64B/66B instance 1. The verbosity of logging can be controlled by setting verbosity parameters in IDF file corresponding to ABFM 64B/66B instance 1 (idffile1)
errorlog0	This file is generated by ABFM 64B/66B and contains the protocol violation errors captured by checker module of ABFM 64B/66B instance 0. Each error is reported via an error code. For a description of error code, refer Table 5-3, page 73.
errorlog1	This file is generated by ABFM 64B/66B and contains the protocol violation errors captured by checker module of ABFM 64B/66B instance 1. Each error is reported via an error code. For a description of error code, refer Table 5-3, page 73.
bfm0_tx_pdu_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the PDU packets transmitted by ABFM 64B/66B instance 0 (2,3)
bfm0_rx_pdu_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the PDU packets received by ABFM 64B/66B instance 0 (2,3)
bfm1_tx_pdu_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the PDU packets transmitted by ABFM 64B/66B instance 1 (2,3)
bfm1_rx_pdu_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the PDU packets received by ABFM 64B/66B instance 1 (2,3)
bfm0_tx_ufc_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the UFC packets transmitted by ABFM 64B/66B instance 0 (2)
bfm0_rx_ufc_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the UFC packets received by ABFM 64B/66B instance 0 (2)
bfm1_tx_ufc_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the UFC packets transmitted by ABFM 64B/66B instance 1(2)
bfm1_rx_ufc_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the UFC packets received by ABFM 64B/66B instance 1(2)
bfm0_tx_userk_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the USER K packets transmitted by ABFM 64B/66B instance 0 (2)
bfm0_rx_userk_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the USER K packets received by ABFM 64B/66B instance 0 (2)
bfm1_tx_userk_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the USER K packets transmitted by ABFM 64B/66B instance 1 (2)

Table 5-2: **Files in Results Directory *(Cont'd)***

Name of the File	Description
bfm1_rx_userk_frames.dat (1)	This file is generated by ABFM 64B/66B and contains the USER K packets received by ABFM 64B/66B instance 1(2)
report.txt	This file is generated by ABFM 64B/66B simulation infrastructure script, run_test.pl, and it contains a summary of the simulation result analysis. For a detailed description of the contents of this file, see "Report File," page 72
idffile0	This file is copied from testcase directory passed as the option value to bfm_testcase argument of script run_test.pl and is used as the IDF file for ABFM 64B/66B instance 0. This file configures the ABFM 64B/66B instance 0 and defines and controls its packet transmission.
Idffile1	This file is copied from testcase directory passed as the option value to partner_testcase argument of script run_test.pl and is used as the IDF file for ABFM 64B/66B instance 1. This file configures the ABFM 64B/66B instance 1 and defines and controls its packet transmission.
sim.do sim.sh (5,7)	Script(s) to be used by simulator while simulating the ABFM 64B/66B - BFM talk mode configuration. Created by ABFM 64B/66B simulation infrastructure script, run_test.pl
mti_wave.do or ncsim_wave.do (5,8)	This file is created by ABFM 64B/66B simulation infrastructure script, run_test.pl only if GUI mode is chosen for simulation. The mode of simulation will be GUI if batch_mode argument is not passed to the run_test.pl script. The simulator uses this script to display waveforms in GUI mode.
abfm_pli_wrapper.v abfm_vhpi_wrapper.vhd abfm_fli_lin_wrapper.vhd abfm_fli_nt_wrapper.vhd	Depending on the language, simulator, and platform choice (4,5,6), one of the language interface wrapper file will be copied from test bench directory. The script internally derives the path for the test bench directory. The wrapper file initializes the ABFM 64B/66B at simulation time zero and handles invocation of user routines provided in the shared library at various events occurring during the simulation. From the point of view of HDL test bench, the HDL language interface wrapper file acts as the ABFM 64B/66B module. The language, simulator, and platform combination and the corresponding language interface wrapper that would be copied is given below: • abfm_pli_wrapper.v: Verilog, any simulator, any platform • abfm_vhpi_wrapper.vhd: VHDL, NCSim, any platform • abfm_fli_lin_wrapper.vhd: VHDL, ModelSim, Linux • abfm_fli_nt_wrapper.vhd: VHDL, ModelSim, Windows
abfm_bfm_tb.v abfm_bfm_tb.vhd	Depending on the language of choice (4), one of these files will be copied from test bench directory. The script internally derives the path for the test bench directory. This file acts as the top-level test bench for the simulation If the language of choice is Verilog, abfm_bfm_tb.v will be copied and if it is VHDL, then, abfm_bfm_tb.vhd will be copied

Table 5-2: **Files in Results Directory *(Cont'd)***

Name of the File	Description
abfm_mti_verilog.so abfm_mti_vhdl.so abfm_ncsim_verilog.so abfm_ncsim_vhdl.so abfm_mti_verilog.dll abfm_mti_vhdl.dll abfm_ncsim_verilog.dll abfm_ncsim_vhdl.dll	Depending on the language, simulator, and platform choice (4,5,6), one of the ABFM 64B/66B shared library will be copied from library directory. The script internally derives the path for the library directory. The language, simulator, and platform combination and the corresponding shared library that would be copied is given below: • abfm_mti_verilog.so: Verilog, ModelSim, Linux • abfm_mti_vhdl.so: VHDL, ModelSim, Linux • abfm_ncsim_verilog.so: Verilog, NCSim, Linux • abfm_ncsim_vhdl.so: VHDL, NCSim, Linux • abfm_mti_verilog.dll: Verilog, ModelSim, Windows • abfm_mti_vhdl.dll: VHDL, ModelSim, Windows • abfm_ncsim_verilog.dll: Verilog, NCSim, Windows • abfm_ncsim_vhdl.dll: VHDL, NCSim, Windows

1. The name of the file used for logging transmitted (or received) packets of particular type could be changed through IDF file. The filenames in this column indicates the default filename in the sample IDF files in ABFM 64B/66B distribution
2. If the packet of particular type is not transmitted (or received), then the file will be generated but will contain no data
3. If ABFM 64B/66B is configured for framing interface, PDU packets are delineated by start-of-frame (#SOF) and end-of-frame (#EOF) symbols. If ABFM 64B/66B is configured for streaming interface, then the PDU packets are logged as a single stream of data
4. Choice of language is indicated by passing the valid language choice as an option value to the 'language' argument of script run_test.pl
5. Choice of simulator is indicated by passing the name of the simulator as an option value to the 'sim' argument of script run_test.pl
6. Choice of platform is indicated by passing the valid platform choice as an option value to the 'platform' argument of script run_test.pl
7. Simulator script, sim.sh is created only if NCSim is chosen as the simulator
8. Only one of the two files is created - mti_wave.do is created if ModelSim is chosen as the simulator and ncsim_wave.do is created if NCSim is chosen as the simulator

Report File

The ABFM 64B/66B top-level test bench terminates the simulation when all the packet requests in IDF files of both instances of ABFM 64B/66B are serviced. Once the simulation terminates, the simulation infrastructure script, run_test.pl invokes a check routine to analyze various log files created by ABFM 64B/66B during the simulation. The script creates a report file report.txt at the end of post processing. The simulation result analysis is summarized in the report file in four sections:

- The first section contains report on analysis of IDF parsing by both instances of ABFM 64B/66B. The simulator log file is analyzed for any possible error reporting on IDF parsing and a failure is reported if any syntax errors are identified in IDF parsing
- The second section contains report on analysis of language interface. The simulator log file is analyzed for any possible error reported from the language interface module of ABFM 64B/66B. Typical error report from language interface module includes port size mismatch between what is defined in test bench and what is internally derived by ABFM 64B/66B data structures
- The third section summarizes the analysis of protocol violations seen by both instances of ABFM 64B/66B instances. An error is reported in this section if either of the two error log files (errorlog0 and errorlog1) is not empty. The error log file contains only the error number of the protocol violation captured by the corresponding instance of ABFM 64B/66B. Table 5-3, page 73 describes the mapping of error code to the corresponding protocol non-compliant behavior seen by the

ABFM 64B/66B. Some of the error-codes are for developer debug only and are presented here for the sake of completion.

Table 5-3: **Error Code Description**

Number	Description
2100	ABFM 64B/66B in Wait For Remote state in simplex mode. In simplex mode, transition to Wait For Remote state is invalid
2112	Invalid USER K block type, that is, USER K block type seen by encoder is not one among 0 to 8
2203	Invalid IDLE control block received, that is, IDLE control block seen by decoder is not one among CC, CB, NR or regular IDLE
2204	Soft error count exceeded threshold
2205	Received a new UFC header even before previous UFC packet was completely received
2206	Received NFC in simplex mode
2207	In a multi-lane configuration, received different NFC PAUSE or NFCs with different XOFF bit status across different lanes in the same databeat
2208	In multi-lane strict alignment configuration, NFC not received on the last lane
2209	In multi-lane strict alignment configuration, UFC header not received on last lane
2210	CC not transmitted by partner on all lanes simultaneously
2211	CC not transmitted by partner with required length
2212	CC not transmitted by partner with required frequency
2213	CB not transmitted by partner on all lanes simultaneously
2214	CB not transmitted by partner with required length
2215	CB not transmitted by partner with required frequency
2216	NR not transmitted by partner on all lanes simultaneously
2217	In multi-lane strict alignment configuration, multiple PDU packets transmitted by partner in the same databeat
2218	In multi-lane strict alignment configuration, multiple UFC packets transmitted by partner in the same databeat
2219	In multi-lane strict alignment configuration, UFC and PDU packets transmitted by partner in the same databeat
2223	In streaming mode, SEP packet transmitted by partner

The fourth and the final section in the report file summarizes the results of analysis of data integrity of packets transmitted and received. For each type of packet, the transmitted packets ABFM 64B/66B instance 0 is compared with the corresponding packet type received by ABFM 64B/66B instance 1, and vice-versa. The transmitted and received packets are logged by ABFM 64B/66B into separate files for each type of packet. Table 5-4, page 74 lists the name of the file pairs used in comparison of data:

Table 5-4: **File Pairs Used for Data Integrity Check**

Log File of Transmit Packets	Log file of Corresponding Received Packets Used in Comparison
bfm0_tx_pdu_frames.dat	bfm1_rx_pdu_frames.dat
bfm0_tx_ufc_frames.dat	bfm1_rx_ufc_frames.dat
bfm0_tx_userk_frames.dat	bfm1_rx_userk_frames.dat
bfm1_tx_pdu_frames.dat	bfm0_rx_pdu_frames.dat
bfm1_tx_ufc_frames.dat	bfm0_rx_ufc_frames.dat
bfm1_tx_userk_frames.dat	bfm0_rx_userk_frames.dat

The name of the file to be used for logging each packet type and direction can be specified in the IDF file of the ABFM 64B/66B instance. However, in the run_test.pl script, these file names are hard coded to the default file names (as shown in Table 5-4) used in the sample IDF files supplied in ABFM 64B/66B distribution. If the default files names are changed through IDF files, then the run_test.pl script should be modified accordingly; else the script will flag error.