# Aurora Bus Functional Model

## *User Guide*

**UG058 (v1.3) May 6, 2005**

XILINX®

# Reference Design and Schematic License

PLEASE READ THIS DOCUMENT CAREFULLY BEFORE USING THE ACCOMPANYING XILINX, INC. AURORA REFERENCE DESIGNS, SCHEMATICS, CODE AND/OR INFORMATION (THE "DESIGNS"). BY USING THE DESIGNS, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE.

You may use the Designs solely in support of your use in developing designs and/or boards for Xilinx programmable logic devices. Access to the Designs is provided only to purchasers of Xilinx programmable logic devices for the purposes set forth herein.

The Designs are provided by Xilinx solely for your reference, for use as-is or as a template to make your own working designs or products. The Designs may be incomplete, and Xilinx does not warrant that the Designs are completed, tested, or will work on their own without revisions. The success of any designs or products you complete or make using the Designs as a starting point is wholly dependent on your design efforts. As provided, Xilinx does not warrant that the Designs or any designs or products you complete or make will provide any given functionality, and all verification must be completed by the customer.

Xilinx specifically disclaims any obligations for technical support and bug fixes, as well as any liability with respect to the Designs, and no contractual obligations are formed either directly or indirectly by use of the Designs.

XILINX SHALL NOT BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION DIRECT, INDIRECT, INCIDENTAL, SPECIAL, RELIANCE OR CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF THE DESIGNS, EVEN IF XILINX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

XILINX IS PROVIDING THE DESIGNS "AS IS" AS A COURTESY TO YOU. BY PROVIDING THE DESIGNS AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THE DESIGNS ARE FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE DESIGNS, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THE DESIGNS ARE FREE FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Xilinx makes no representation that the Designs or any designs or products you complete or make will provide the functionality you are looking for, or that they are appropriate for any given use. Xilinx does not warrant that the Designs are error-free, nor does Xilinx make any other representations or warranties, whether express or implied, including without limitation implied warranties of merchantability or fitness for a particular purpose. The Designs are not covered by any other license or agreement you may have with Xilinx.

The Designs are the copyrighted information of Xilinx. You may not reproduce, transmit or otherwise copy the Designs by any means for any purpose not set forth in this license, without the prior written permission of Xilinx.

You agree that you will comply with all applicable governmental export rules and regulations, and that you will not export or reexport the Designs in any form without the appropriate government licenses.

XILINX, INC., 2100 Logic Drive, San Jose, California 95124

# Revision History

The following table shows the revision history for this document.

| Date | Version | Revision |
|---|---|---|
| 10/17/03 | 1.0 | Initial Xilinx release. |
| 06/08/04 | 1.1 | Added new text and figures to Protocol Conformance section;<br>Added new arguments to Table 2-1;<br>Miscellaneous text and figure edits for clarity. |
| 10/27/04 | 1.2 | Added Appendix A: Aurora BFM: Simplex with Serial Interface;<br>Miscellaneous non-technical edits for clarity. |
| 05/06/05 | 1.3 | Added Schedule of Figures (SOF) and Schedule of Tables (SOT);<br>Added more function calls in Chapter 2: PLI Interface and Test Bench Integration;<br>Added Appendix B: Aurora BFM: Parallel Interface. |

# Table of Contents

## Appendix A: Simplex with a Serial Interface



## Appendix B: Aurora BFM: Parallel Interface

# Schedule of Figures

## Chapter 1: Aurora Bus Functional Model



## Chapter 2: PLI Interface and Test Bench Integration

## Chapter 3: FLI Interface

## Appendix A: Simplex with a Serial Interface



## Appendix B: Aurora BFM: Parallel Interface

# Schedule of Tables

# About This User Guide

This document describes an Aurora bus functional model (ABFM). The ABFM models the behavior of the Aurora protocol and can be used to generate stimulus for and to monitor the response of an Aurora interface design, which is referred to as the device under test (DUT).

## User Guide Contents

This user guide contains the following chapters:

Chapter 1, "Aurora Bus Functional Model" is an overview of the ABFM, providing details about the Aurora application model, user application interface, protocol conformance, clocking scheme, and error injection.

Chapter 2, "PLI Interface and Test Bench Integration" describes the details of the programming language interface (PLI) and its integration into the test bench.

Chapter 3, "FLI Interface" describes the foreign language interface (FLI) that provides the ability to interface with the BFM shared library from a VHDL environment. It can be considered as a VHDL equivalent of the PLI.

Appendix A, "Simplex with a Serial Interface" describes the specification of the Aurora simplex BFM which supports the verification of Aurora simplex designs.

Appendix B, "Aurora BFM: Parallel Interface" describes the interface to the Aurora BFM that enables the user to hook up the Aurora DUT directly to the BFM bypassing the MGT functionality.

## Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this website. You can also directly access these resources using the provided URLs.

| Resource | Description/URL |
|---|---|
| Tutorials | Tutorials covering Xilinx design flows, from design entry to verification and debugging<br>http://support.xilinx.com/support/techsup/tutorials/index.htm |
| Answer Browser | Database of Xilinx solution records<br>http://support.xilinx.com/xlnx/xil_ans_browser.jsp |

| Resource | Description/URL |
|---|---|
| Application Notes | Descriptions of device-specific design techniques and approaches<br>http://support.xilinx.com/apps/appsweb.htm |
| Data Sheets | Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging<br>http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp |
| Problem Solvers | Interactive tools that allow you to troubleshoot your design issues<br>http://support.xilinx.com/support/troubleshoot/psolvers.htm |
| Tech Tips | Latest news, design tips, and patch information for the Xilinx design environment<br>http://www.support.xilinx.com/xlnx/xil_tt_home.jsp |

# 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`* |

| Convention | Meaning or Use | Example |
|---|---|---|
| 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}` |
| Vertical ellipsis<br>.<br>.<br>. | Repetitive material that has been omitted | `IOB #1: Name = QOUT'`<br>`IOB #2: Name = CLKIN'`<br>.<br>.<br>. |
| 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 file or in another file in the current document | See the section "Additional Resources" for details.<br>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-II Handbook*. |
| Blue, underlined text | Hyperlink to a website (URL) | Go to http://www.xilinx.com for the latest speed files. |

XILINX®

Chapter 1

# Aurora Bus Functional Model

## Introduction

This document describes an Aurora bus functional model (ABFM). The ABFM models the behavior of the Aurora protocol and can be used to generate stimulus for and to monitor the response of an Aurora interface design, which is referred to as the device under test (DUT).

The ABFM provides parametrization of the protocol parameters (for example, the number of lanes) and that can be used to test any implementation of the Aurora protocol with little overhead. The ABFM provides flexibility, a *clean room* implementation of Aurora, and improved performance over using another Aurora design to verify the DUT.

The ABFM can be easily integrated (described in "Test Bench Integration," page 39) into an existing verification environment specifically designed to test the DUT. Once the ABFM is integrated into the verification environment, it can communicate with the DUT using a programming language interface (PLI) for Verilog environments or a foreign language interface (FLI) for VHDL+ModelSim environment. Both PLI and FLI contain transaction-based calls that are used to establish communication between the DUT and ABFM.

The ABFM is compatible with multiple simulators and multiple platforms, and supports both Verilog and VHDL environments. The combinations available are:

- **ModelSim:** SunOS, Windows (Win32), and Linux (VHDL only)
- **NCSIIM:** SunOS, Windows (Win32) and Linux (VHDL only)
- **VCS:** SunOS (Verilog only)

The ABFM package deliverables include:

- A test environment containing test benches (simulates two BFM's communicating with each other in a 4-lane configuration)
- Shared libraries containing PLI routines and interface (for Verilog environments)
- Shared libraries containing FLI routines and interface (for VHDL environments). This facilitates seamless communication between the three components of the package: ABFM, test benches, and the DUT

Figure 1-1 shows a block diagram of the ABFM environment.

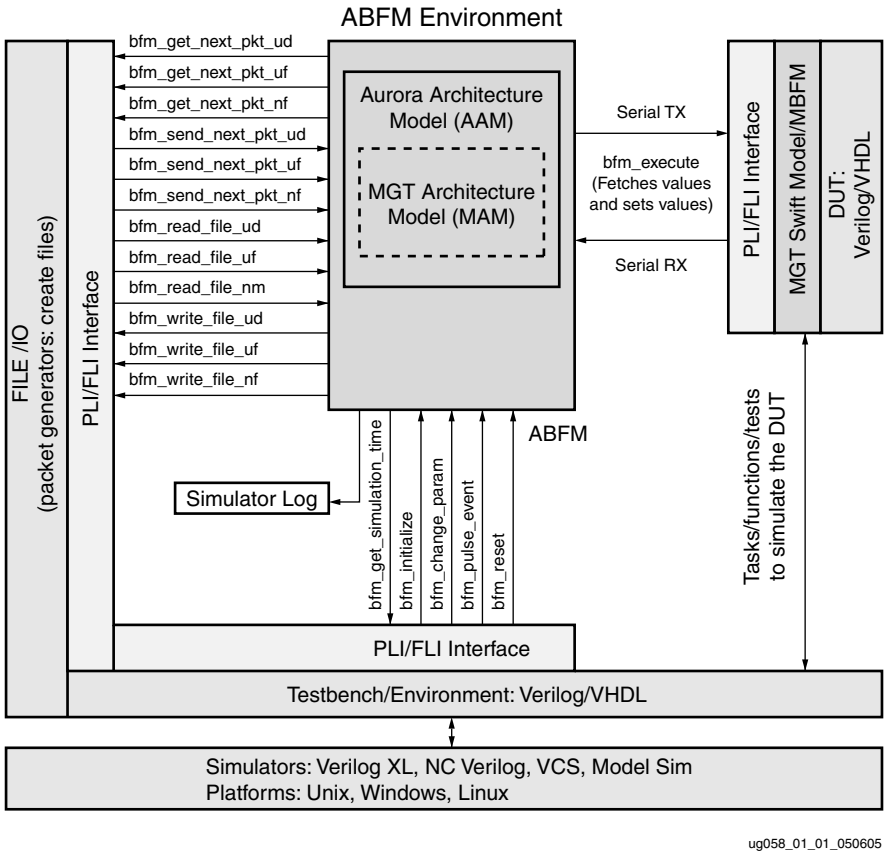


*Figure 1-1:* **ABFM Environment**

# Architecture

The ABFM architecture can be broadly categorized into the following sections:

- "Aurora Architecture Model," page 17
- "User Application Interface," page 20
- "Protocol Conformance," page 21
- "Clocking Scheme," page 32
- "Error Injection," page 33

The Aurora architecture model (AAM) contains C models of the Aurora protocol and a RocketIO™ multi-gigabit transceiver (MGT). The AAM communicates with the test benches through a PLI interface (for Verilog environments) or an FLI interface (for VHDL environments). The PLI and FLI interfaces contain routines and functions that can be called from the test bench using system calls (refer to Chapter 2, "PLI Interface and Test Bench Integration" and Chapter 3, "FLI Interface" for more details).

Table 1-1 shows the PLI and FLI system calls. Note the difference in syntax between the two.

*Table 1-1:* **PLI and FLI System Calls**

| PLI System Call | FLI System Call |
|---|---|
| $bfm_initialize | bfm_initialize |
| $bfm_execute | bfm_execute |
| $bfm_read_file_ud | bfm_read_file_ud |
| $bfm_read_file_uf | bfm_read_file_uf |
| $bfm_read_file_nf | bfm_read_file_nf |
| $bfm_write_file_ud | bfm_write_file_ud |
| $bfm_write_file_uf | bfm_write_file_uf |
| $bfm_write_file_nf | bfm_write_file_nf |
| $bfm_send_next_pkt_ud | bfm_send_next_pkt_ud |
| $bfm_send_next_pkt_uf | bfm_send_next_pkt_uf |
| $bfm_send_next_pkt_nf | bfm_send_next_pkt_nf |
| $bfm_get_next_pkt_ud | bfm_get_next_pkt_ud |
| $bfm_get_next_pkt_uf | bfm_get_next_pkt_uf |
| $bfm_get_next_pkt_nf | bfm_get_next_pkt_nf |
| $bfm_change_param | bfm_change_param |
| $bfm_reset | bfm_reset |
| $bfm_get_simulation_time | bfm_get_simulation_time |
| $bfm_pulse_event | bfm_pulse_event |

# Aurora Architecture Model

The Aurora architecture model (AAM) emulates the behavior of the Aurora protocol in C. There is a very close relationship between the AAM and the actual implementation of the design. The AAM is derived from the *Aurora Protocol Specification* (SP006) http://www.xilinx.com/aurora and is used as a *golden model* to verify the protocol conformance of the DUT.

The AAM's goal is to only check the Aurora implementation, not the SERDES operation in the real world or the implementation of the user interface in the DUT. The AAM contains functions that model the individual components of the Aurora protocol.

The AAM is not meant to be cycle-accurate internally. The latencies seen by the DUT are not modeled in the AAM. Any timing-related behavior (for example, pipelining data to meet timing) is not modeled in the AAM. However, the MGT architecture model (MAM) contains hooks to skew lanes to model line characteristics.

Figure 1-2, page 18 describes a test environment example that shows how the ABFM can be used in loopback mode to test the DUT. Packets from a file can be loaded into the ABFM by making system calls in the wrapper for ABFM blocks that are executed through the PLI

interface block. The ABFM then transmits the data across the serial TX port. This data is received by the DUT through the wrapper for ABFM and PLI interface blocks. The DUT processes the data and loops back the received packets through the serial RX port. This data is processed by the ABFM which dumps the packets into a file. The user can verify the results by comparing the files from which the packets were sent and the files into which the packets were received.

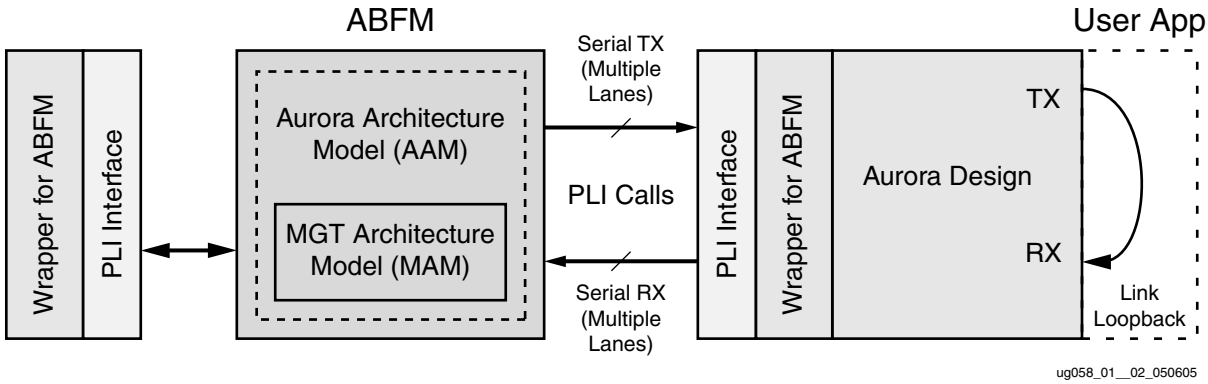


*Figure 1-2:* **ABFM Test Environment Example**

Figure 1-3, page 19 shows the functional blocks in the ABFM. The functions modeled in the AAM are listed as follows:

- **Link Encap**: Encapsulates each packet with delimiters and adds padding if necessary to create an even-sized encapsulated packet.
- **Src Wait En**: Forces the transmit scheduler to generate idle sequences. This emulates breaks in the data stream received by an Aurora receiver from the source device.
- **TX Scheduler**: Evaluates the next command in the pipeline based on commands currently pending and the priority order of command types. It causes the appropriate symbols and control words to be sent to the lane striper.
- **Lane Striper**: Receives data from the transmit scheduler and stripes it across multiple lanes.
- **Lane Distributor**: Receives the two or four bytes for each lane from lane striper and sends one byte after another to the 8B/10B encode function.
- **8B/10B Encode**: Encodes the received data and generates/monitors the running disparity.
- **Serializer**: Sends out the encoded data one bit at a time.
- **Deserializer**: Accumulates serial data received on the line, one bit at time.
- **Comma Align**: Performs comma alignment and locks to the code boundary of the incoming data.
- **8B/10B Decode**: Decodes the received data, performs running disparity checks, and sends the data/control symbols to lane destriper.
- **Lane Assembler**: Receives one decoded byte after another from 8B/10B decoder, assembles enough to prepare two or four bytes for the lane and sends it to the lane destriper.
- **Lane Destriper**: Assembles data across all lanes and sends data to the receiver distributor.

- **RX Distributor**: Segregates incoming data into *user protocol data unit* (user PDU), *user flow control* (UFC) PDU, *native flow control* (NFC) PDU; drops idles and *clock compensation* (CC) sequences; sends command to transmit scheduler if channel partner NFC PDU is received.
- **Link Decap**: Drops encapsulation delimiters and padding.
- **Ch Init Verif**: Performs channel initialization, bonding, and verification for all lanes; overrides data being sent from transmit scheduler until successful completion of channel initialization.

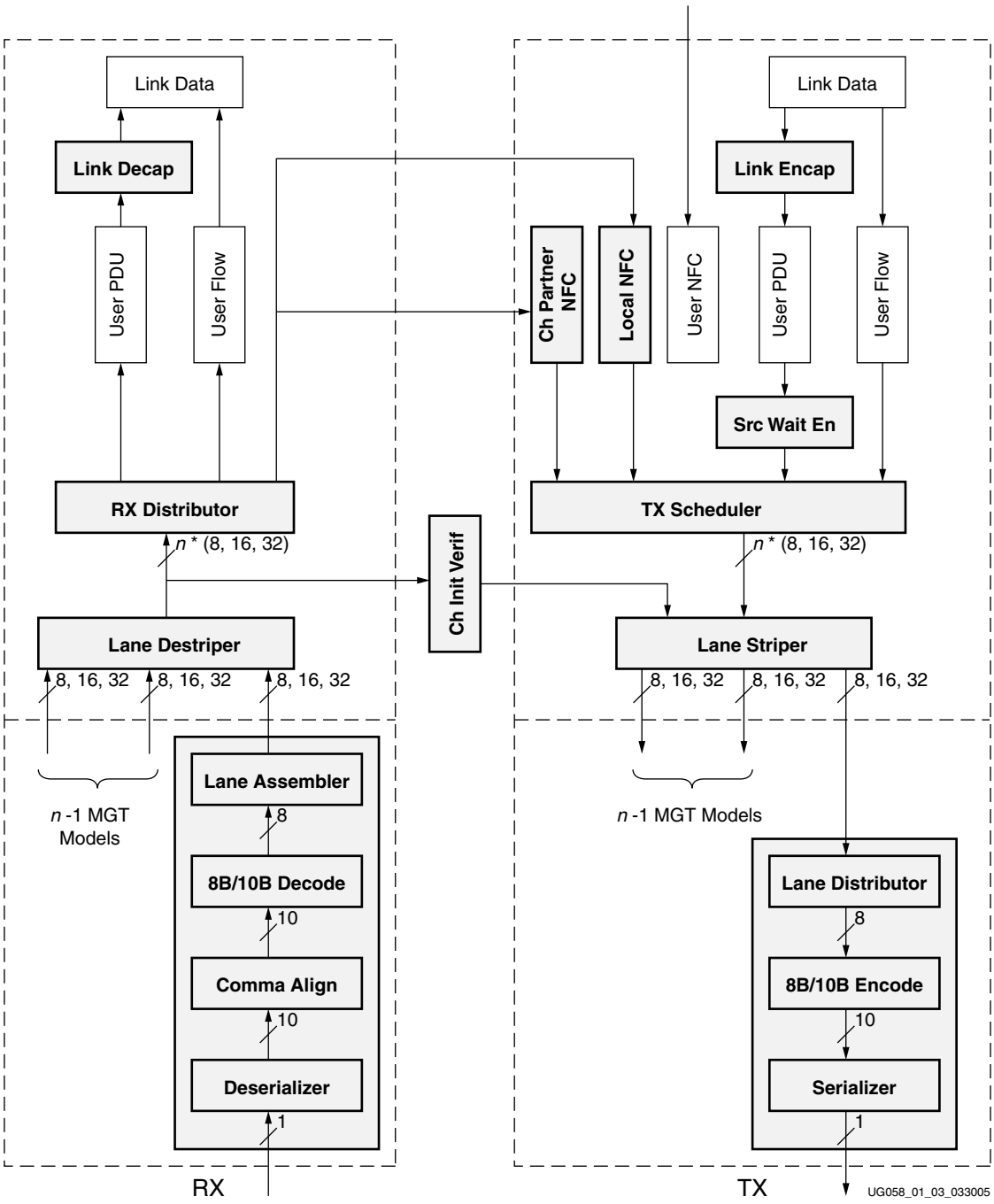


*Figure 1-3:* **ABFM Functions**

# User Application Interface

The ABFM uses file transfer techniques to model the user application interface. This interface enables the user to send and receive user PDU, UFC PDU, and NFC PDU packets to and from the ABFM. The packets are generated by a pre-processor (described in "Packet Generator," page 50). The packet format uses simple delimiters such as #SOF and #EOF to describe packets. By default, each line has eight bytes (emulating a 64-bit interface).

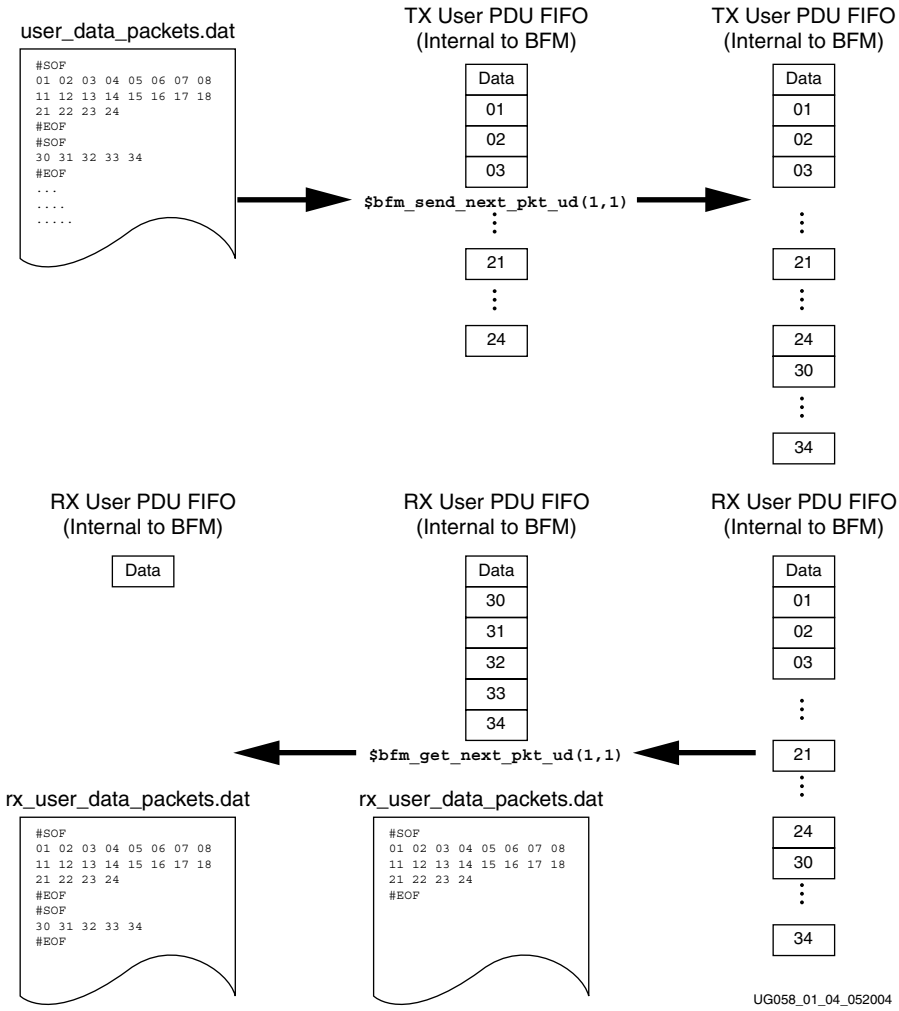


*Figure 1-4:* **Sending/Receiving User PDU Packets to/from the ABFM**

Several options are provided to modify the packet formats. These are described in detail in "Packet Generator," page 50. Internal FIFOs (three types: user PDU, UFC, and NFC) in the ABFM are created for both TX path (TX FIFOs) and RX path (RX FIFOs) of the ABFM. Figure 1-4 is an example showing how user PDU packets are sent to the ABFM from a file and how they are retrieved from the ABFM into a file.

In this particular example, the file that stores packets to be transmitted is called `tx_user_data_packets.dat` and the file that receives packets is called `rx_user_data_packets.dat`. First, the user needs to open these files and associate them to the FIFOs in the BFM. This is done using the following system calls in the test bench (refer to Chapter 2, "PLI Interface and Test Bench Integration"):

- $bfm_read_file_ud(id,"user_data_packets.dat"), which associates the file user_data_packets.dat to TX user PDU FIFO
- $bfm_write_file_ud(id,"rx_user_data_packets.dat"), which associates the file rx_user_data_packets.dat to RX user PDU FIFO

Once the files are opened and associated with the FIFOs, the user can send or receive any number of packets from the internal FIFOs using the following system calls:

- $bfm_send_next_pkt_ud(id,1);
- $bfm_get_next_pkt_ud(id,1);

The example shows how two packets are loaded one after the other into the FIFO showing the state of the FIFO after each system call. It also shows how two packets are drained from the FIFO into a file showing the status of the FIFO and the file after each system call. The transmit scheduler in the ABFM monitors all the TX FIFOs to determine what needs to be sent on the channel. The RX distributor in the ABFM distributes the received data from the channel into the appropriate RX FIFOs. This provides a simple mechanism for emulating the user application.

# Protocol Conformance

Several protocol conformance checks are built into the ABFM. The checks are classified into two categories:

- "Continuous Conformance Checks"
- "Prepackaged Conformance Checks," page 23

## Continuous Conformance Checks

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 ABFM accumulates error counts for each individual error type and stores them internally in a data structure.

The error log can be dumped into the simulator's output by using a system call (refer to Chapter 2, "PLI Interface and Test Bench Integration."):

- $bfm_change_param(id,"log_errors", 1);

The user can make the call as many times as desired. The various checks and the resulting error categorization made by the ABFM are listed below.

The following errors are logged on a *per lane* basis:

- **Xs, Zs**: The ABFM checks Xs and Zs on the serial bits.
- **BIT=BIT_N**: The ABFM checks if the serial bit (rx_p) and its complement (rx_n) are truly complements of each other.
- **NOT IN TABLE**: Accumulation of the *not in table* errors received from the MGT model in the ABFM.
- **WRONG RUNDISP**: Accumulation of the *rundisp err* errors received from the MGT model in the ABFM.

The following errors are logged on a *per channel* basis:

- **INVALID SYMBOLS**: Incoming control symbols are checked for Aurora conformance. Valid 8B/10B control symbols which do not fall into the Aurora control symbol space (contains all control symbols recognized by Aurora) will be treated as an error.
- **/SCP/ ERRORS**: If one of $/SCP/_1$ or $/SCP/_2$ is corrupt, then it is treated as an /SCP/ error. Note that a bad $/SCP/_1$ leads to the /SCP/ not being recognized and results in the packet being dropped.
- **/ECP/ ERRORS**: If one of $/ECP/_1$ or $/ECP/_2$ is corrupt, then it is treated as an /ECP/ error. Note that a bad /ECP/ leads to the /ECP/ not being recognized. This leads to unpredictable results.
- **PDU ERRORS**: If in the middle of a packet, a control symbol which is neither an idle, CC, UFC header, NFC header, or PAD header is received, it is treated as a PDU error.
- **EMBEDDED PDU ERRORS**: This check looks for missing /SCP/, /ECP/ pairs. Every /SCP/ needs to be followed by an encounter of /ECP/ before another /SCP/ can be encountered. Similarly, an /ECP/ can be encountered only if an /SCP/ has already been encountered. Any other combination is treated as an error.
- **PAD ERRORS**: If PAD symbol is found in the even position in a user PDU, it is treated as an error.
- **ODD PDU ERRORS**: If the $/ECP/_1$ is found in an odd position, it indicates a PDU of odd length and treated as an error.
- **UFC SIZE ERROR**: If the extracted UFC PDU (encounter of /SCP/ can terminate the extraction of the UFC PDU) is not of same length as the value in the size field, it is treated as an error.
- **UFC DATA ERROR**: If a control symbol is found before the required number of symbols based on the value of size field are received, then it is treated as an error.
- **NFC ERRORS**: If the symbol following the /SNF/ symbol is a control symbol, it is treated as an error.
- **IDLE ERRORS**: This checks if the idles received across lanes in the channel are identical. Note that the idles received in each lane are dumped into a file. The user can post process it to do further analysis.
- **CC LENGTH ERRORS**: If the CCs received contiguously are not of the required length, it is treated as an error.
- **CC FREQUENCY ERRORS**: If the gaps between successive CC occurrences is greater than the required gap, it is treated as an error. Note that if the gap is smaller, it is not treated as an error.
- **ZERO PDU ERRORS**: If a /SCP/ is immediately followed by /ECP/, it indicates a PDU of size zero. This is treated as an error.

In addition to the error logging, incoming data that does not conform to Aurora protocol leads to dropped packets, merged packets, and so on. These can be easily verified by comparing the files which contain packets that were transmitted and files which contain packets that were received.

An initialization monitor signal `init_monitor[0:no_of_lanes+3]` is provided in the ABFM to monitor the status of the ABFM state machines. The mapping of the bits in the signal are described below:

- 0:no_of_lanes-1: Lane up indicators for each lane
- no_of_lanes: Successful completion of channel bonding
- no_of_lanes+1: Successful completion of channel verification
- no_of_lanes+2: Channel up

## Prepackaged Conformance Checks

Prepackaged conformance checks contain algorithms that generate specific pattern sequences and rely on an *echo* configuration (except for initialization checks) of the channel partner (the DUT) to verify protocol conformance. In this check, the ABFM needs to control the data it receives from the channel partner. Therefore, it requires the channel partner to loopback the data received from the ABFM through an external mechanism.

Any violations of the protocol compliance can be easily detected using prepackaged checks. The following prepackaged tests are supported by the ABFM:

- "Initialization Phase," page 24
- "Channel Up Phase," page 28

***Note:*** The prepackaged checks can be used to verify the quality of the Aurora implementation under test, but do not guarantee complete Aurora conformance. The user is recommended to use the ABFM as a verification tool and further create external tests in the test bench to fully verify the Aurora implementation. The checks are not replacements for verification coverage.

## Initialization Phase

In this phase, the ABFM controls the initialization of the channel partner by injecting errors into the ABFM's transmit path at pre-determined points. It forces the channel partner to exercise all arcs of the lane initialization, channel bonding, and channel verification state machines. this test does not require the channel partner to be configured in *echo* mode.

The algorithms for the initialization phase are shown in Figure 1-5, page 25, Figure 1-6, page 26, and Figure 1-7, page 27. The algorithms send error messages to the simulator's log whenever a check fails. The error messages are provided with error ids (marked in the flow charts for the algorithms) to identify the exact state in which the check failed.

The initialization phase conformance check can be initiated with the following call at start of simulation:

- $bfm_change_param(id,"chk_init_conf","1") /*turns on initialization conformance only*/

  OR

- $bfm_change_param(id,"chk_conf","1")/*turns on initialization and channel up conformance*/

The overall status of the initialization phase test can be checked by the following PLI call after the channels for the DUT and the ABFM are up:

- $bfm_change_param (id, "log_errors","1")

***Note:*** The response time for the DUT is related to the latency of the implementation; therefore, the parameters used in the testing engine need to factor them. The latency number of the DUT needs to be conveyed to the ABFM using the PLI call $bfm_change_param(id,"system_latency","value"), where "value" is an integer value of the system latency of the channel partner. The user needs to pick a conservative number for "system_latency" to ensure proper testing.

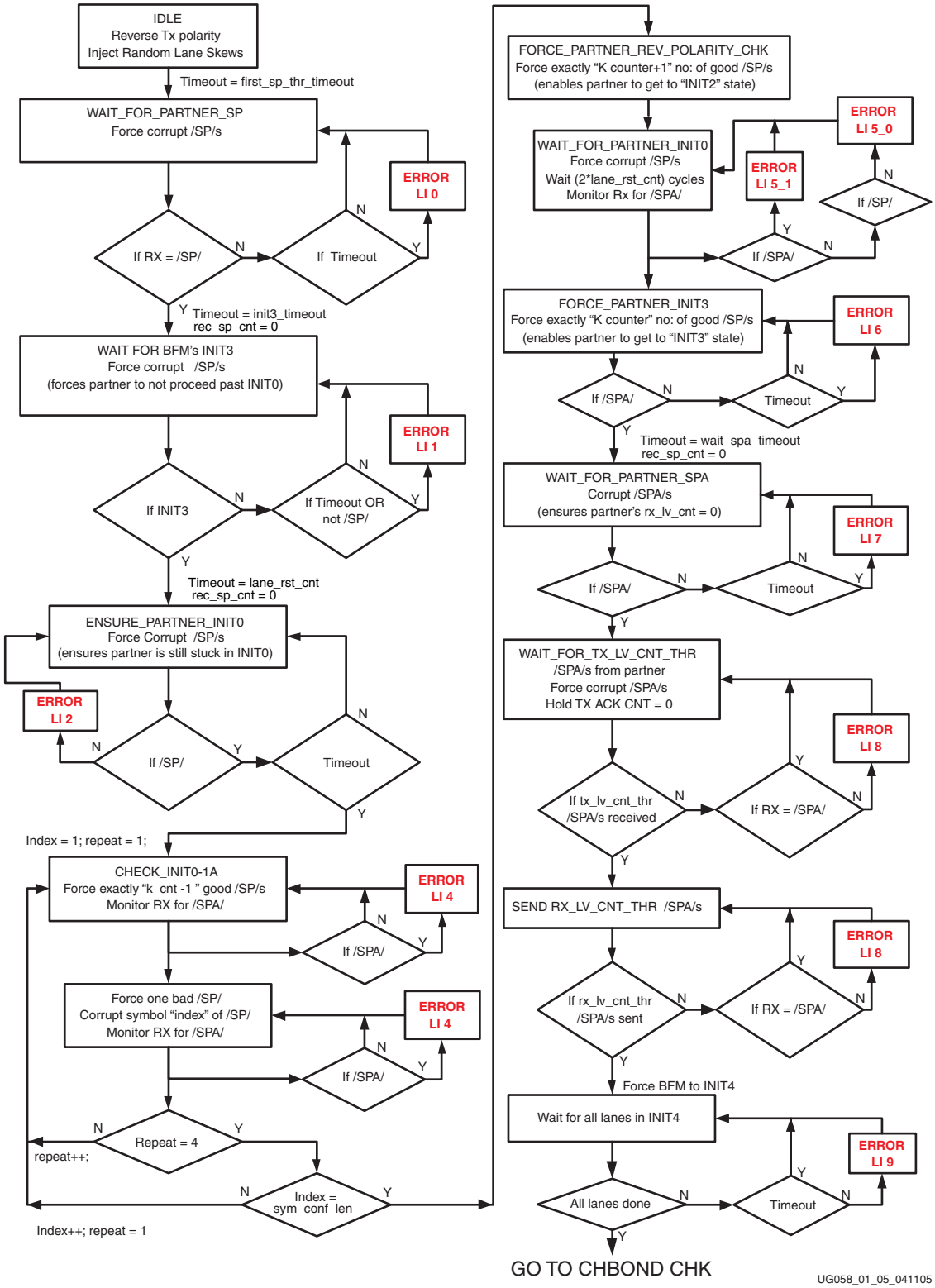


*Figure 1-5:* **Lane Initialization Checks**

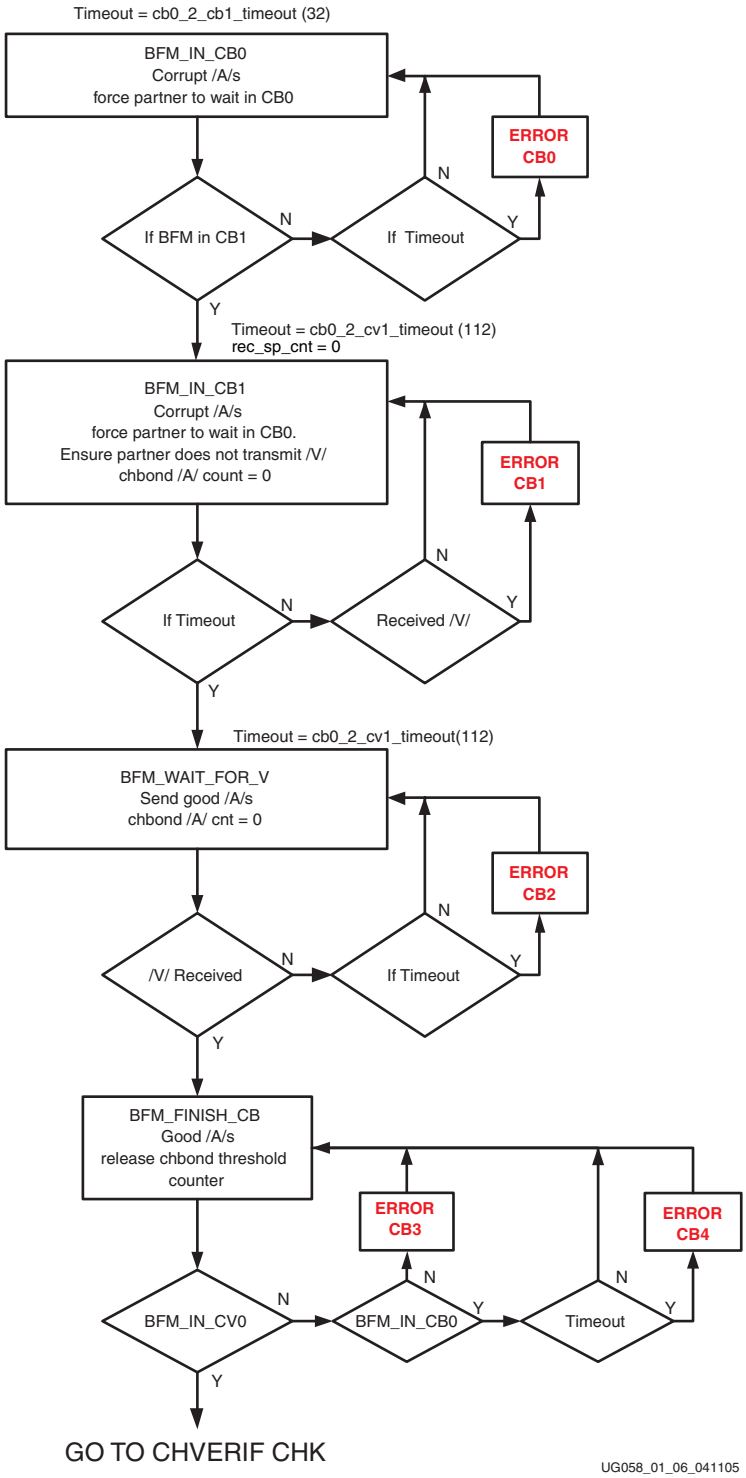


*Figure 1-6:* **Channel Bond Checks**

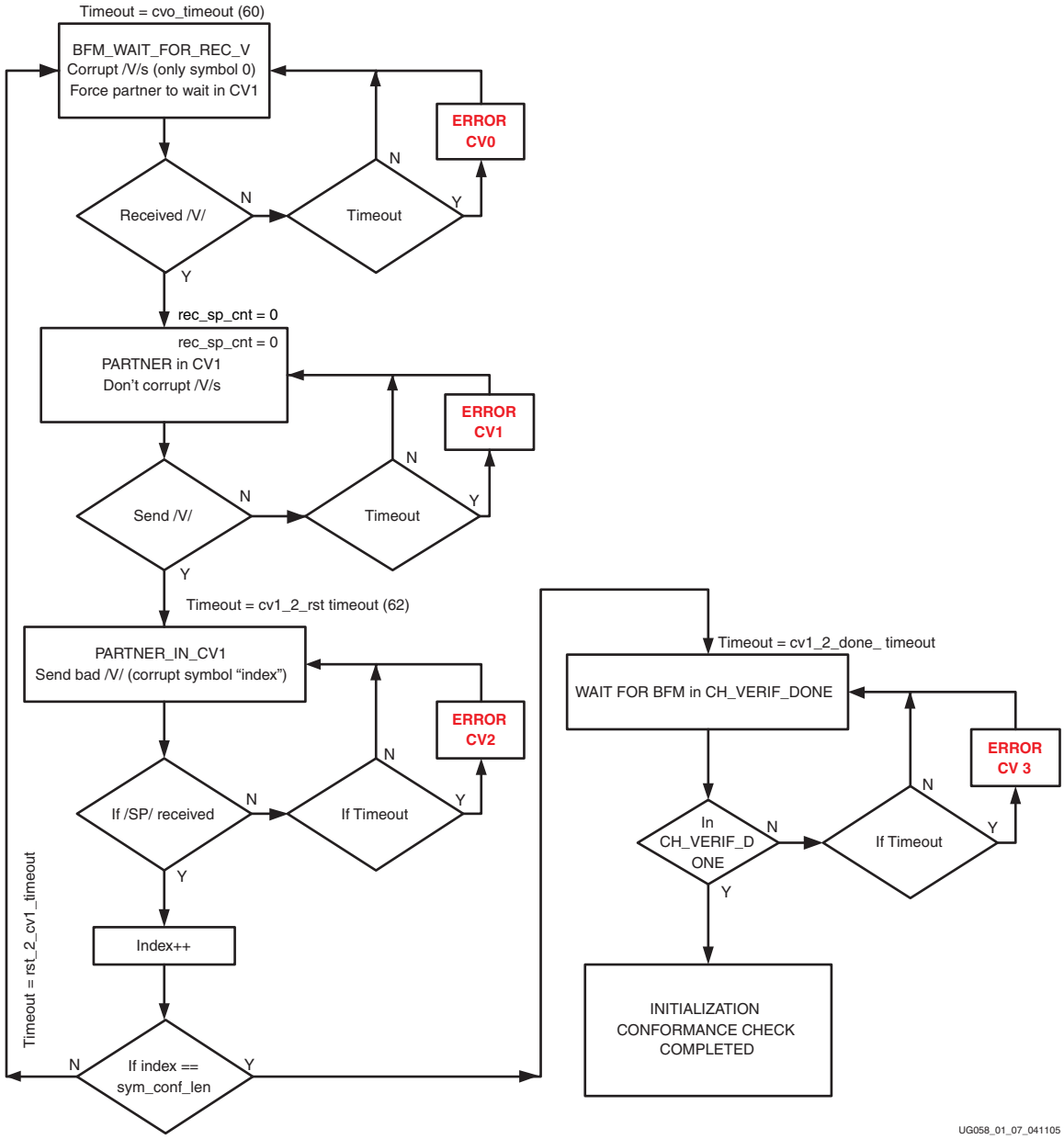


*Figure 1-7:* **Channel Verification Checks**

## Channel Up Phase

In this phase, the DUT needs to be configured in an *echo* mode which loops back PDUs from its RX path to TX path. The ABFM sends various combinations of user, UFC, and NFC PDUs (embedded and non-embedded) after the ABFM's channel is up and checks the received PDUs to determine protocol conformance. It also performs soft error algorithm checks. The algorithms send error messages to the simulator's log whenever a check fails. The error messages are provided with error ids (marked in the flow charts for the algorithms) to identify the exact state in which the check failed. The initialization phase conformance check can be initiated with the following call at start of simulation:

- $bfm_change_param(id,"chk_cup_conf","1") /*turns on channel up conformance only*/

  OR

- $bfm_change_param(id,"chk_conf","1")/*turns on initialization and channel up conformance*/

The overall status of the initialization phase test can be checked by the following PLI call after the channels for the DUT and the ABFM are up:

- $bfm_change_param (id, "log_errors","1")

The channel up phase and soft error conformance phase are shown in Figure 1-8, page 29, Figure 1-9, page 30, and Figure 1-10, page 31. Two additional interface ports `conf_monitor[0:7]` and `conf_result[0:no_of_lanes+5]` between the ABFM and test bench is provided to monitor protocol conformance. This is provided through an extension of the ports in the $bfm_initialize(…) call. The mapping of the individual bits in conf_monitor are described below:

- 0: chk_conf /*conformance check enable for all modes*/
- 1: chk_conf_done /*conformance checks for all modes is complete */
- 2: chk_init /*conformance check enable for initialization phase only*/
- 3: chk_init_conf_done /*conformance check for initialization phase is complete*/
- 4: chk_cup_conf /*conformance check enable for channel up phase*/
- 5: chk_cup_conf_done /*conformance check for channel up phase is complete*/
- 6: chk_softerr_conf /*conformance check enable for softerror algorithm */
- 7: chk_softerr_conf_done /*conformance check for softerror algorithm is complete*/

The mapping of the individual bits in conf_result are described below:

- 0: channel conformance result
- 1: check initialization result
- 2: check channel up conformance result
- 3: no_of_lanes+2: Lane initialization result for each lane
- no_of_lanes+3: Channel bonding result
- no_of_lanes+4: Channel verification result
- no_of_lanes+5: User PDU mode result
- no_of_lanes+6: UFC mode result
- no_of_lanes+7: NFC mode result
- no_of_lanes+8: Soft error check result
- The user can monitor the above signals to detect conformance failures instantly.

***Note:*** A continuous conformance block always monitors the received data to check SCPs, ECPs, SUFs, SNFs, CCs, idles, etc. The user should always check the status of the continuous conformance checks.

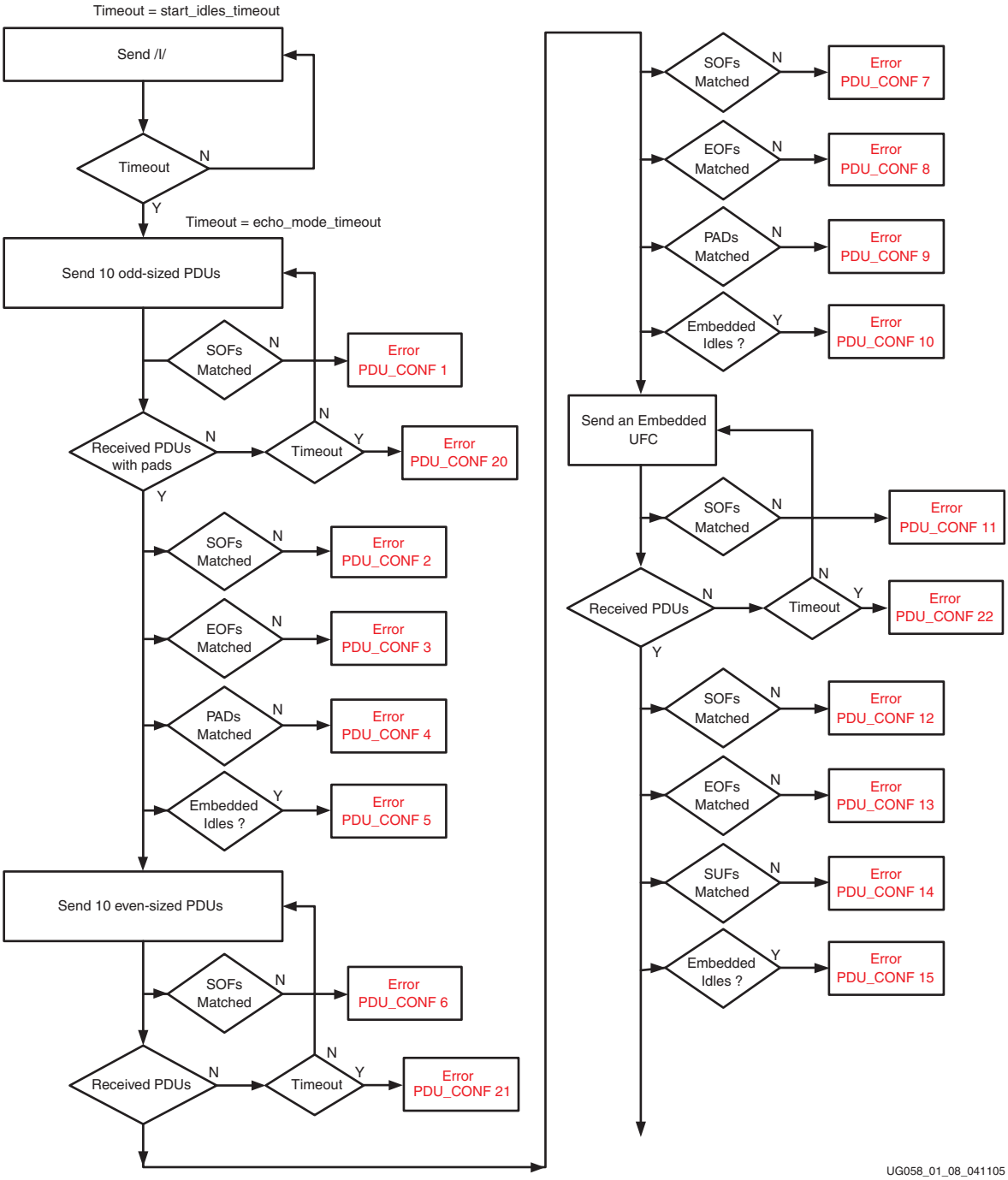


*Figure 1-8:* **Channel Up Phase (Part 1 of 2)**

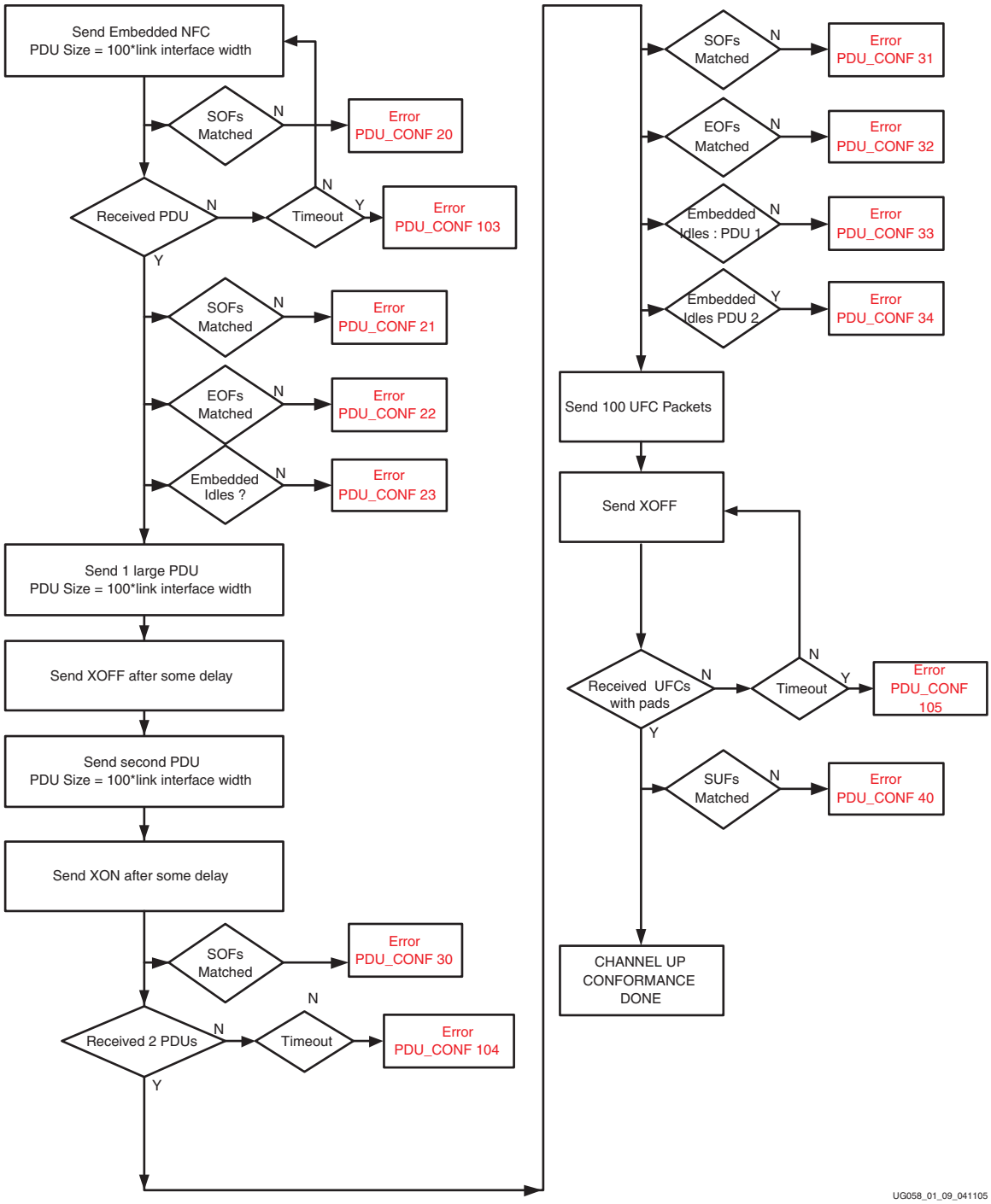


*Figure 1-9:* **Channel Up Phase (Part 2 of 2)**

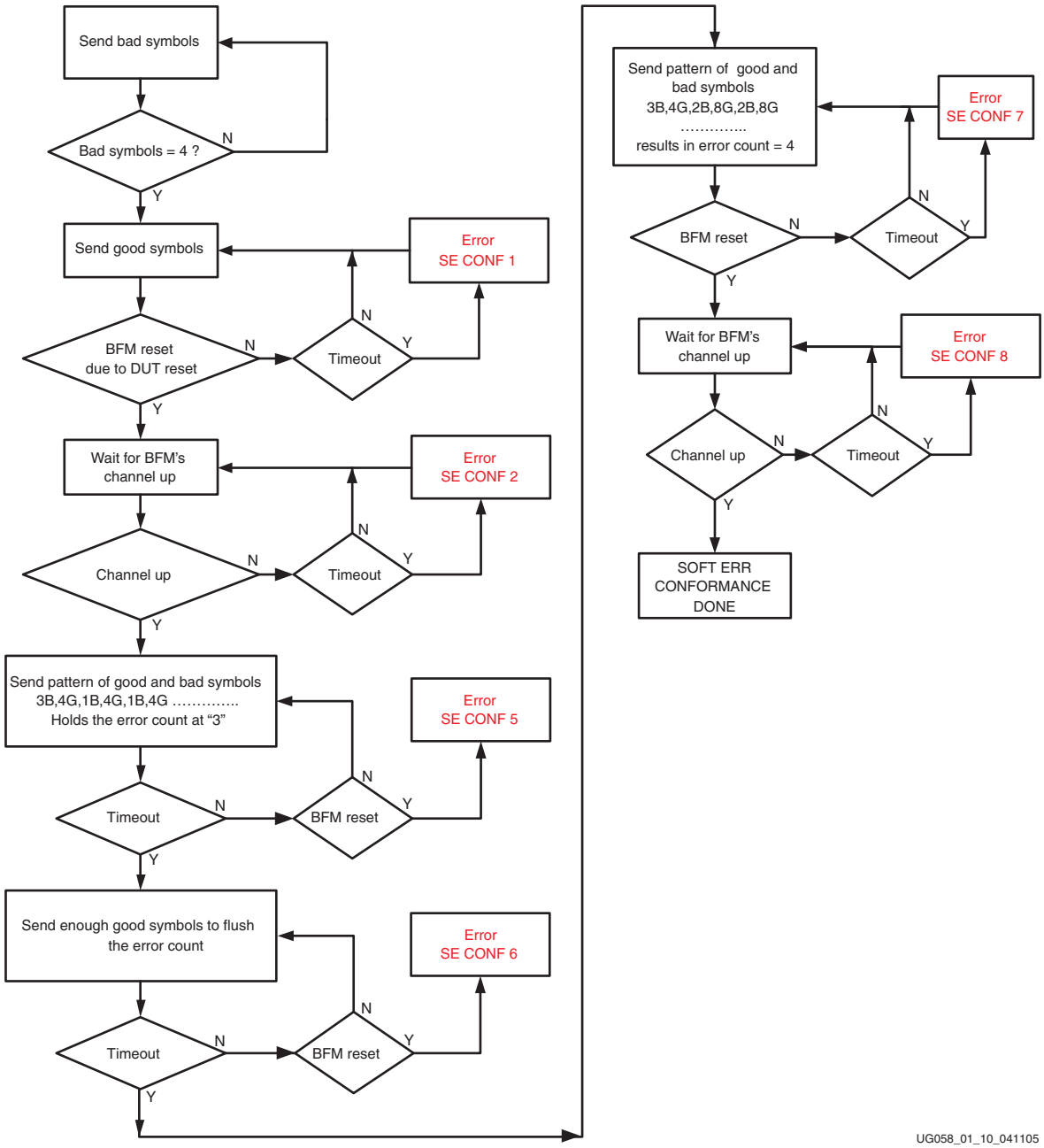


*Figure 1-10:* **Soft Error Conformance**

# Clocking Scheme

The ABFM communicates with the DUT through a serial interface. The ABFM requires a clock input that matches the serial data rate of the DUT. The serial clock, however, is internally generated and inaccessible outside the DUT. Therefore, a serial clock must be generated to match the serial data rate in the test bench. This serial clock should be derived from the clock input to the DUT.

The RX path of the DUT recovers the clock from the incoming data for its internal clocking. The ABFM cannot perform clock recovery, so it expects data and a clock (synchronous to the data) to be provided by the test bench. This clock is used to clock the serial RX path of ABFM. Data from the serial TX path of the ABFM is transmitted synchronous to the same clock. Hence, the ABFM has both TX and RX paths running on the same clock.

The internal functions of the ABFM run on emulated clocks derived from the serial RX clock. The different clock zones in the ABFM are shown in Figure 1-11 for reference. Clock emulation is performed by controlling how often a function is called in relation to the number of clock cycles in the simulation. A function called every 10 clocks runs on a divide-by-10 clock. Similarly a divide-by-20 clock can be generated. A divide-by-*x* clock is emulated by calling a function every *x* clock cycles. The clock zoning of ABFM is for reference only and should not be confused with the internal clocking of the DUT.

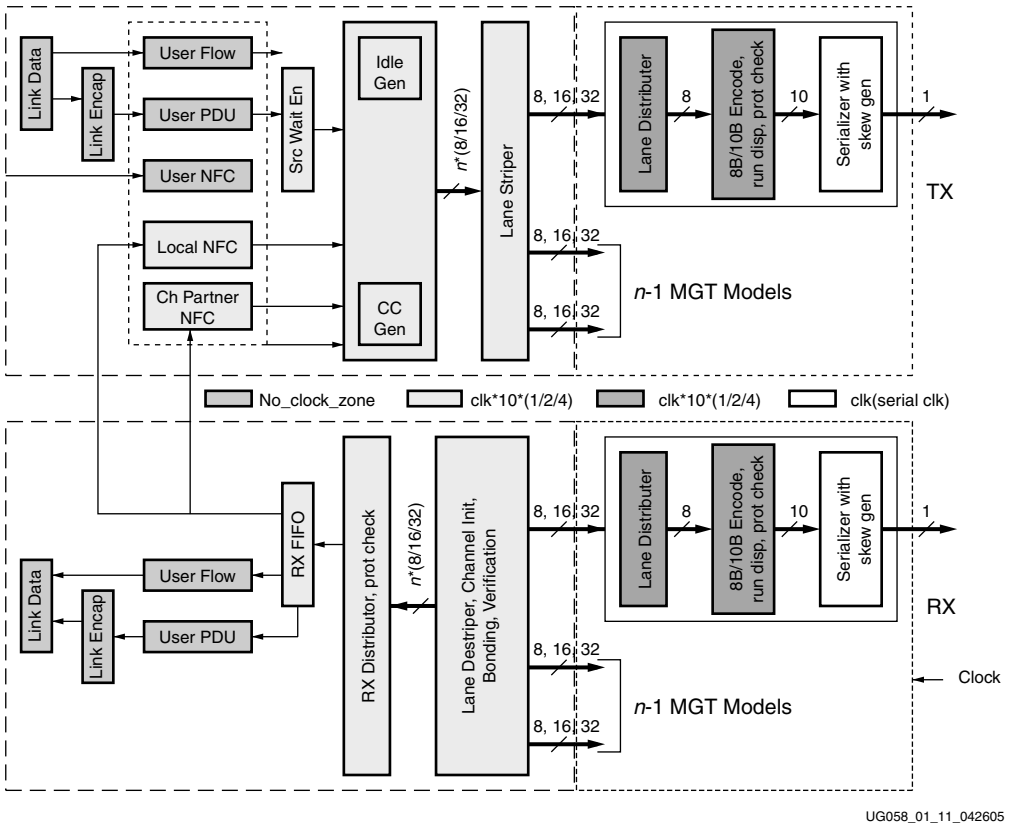


*Figure 1-11:* **Clocking Scheme**

# Error Injection

Error injection capabilities are provided in the ABFM to manipulate the data on the TX path of ABFM. These capabilities are classified into two categories:

- "Symbol Corruption," page 34
- "Symbol Deletion," page 36

The user can use this capability to inject errors on the TX path of ABFM and force the RX path of the DUT to process errors and stress corner cases. The $bfm_change_param call (described in "$bfm_change_param(id,param1,val,param2..,paramn,valn)," page 45) is used to inject errors. The specific call and the parameters are described in BNF format as shown in Figure 1-12. A tabular format is also provided in Table 2-1, page 45.

```
$bfm_change_param("error_inject",error_descriptor)
error_descriptor :=     "error_type error_pattern" |
                        "error_type error_pattern error_size" |
                        "error_type error_pattern error_size symbol_select" |
error_type := scp | ecp | suf | snf | idle_a | idle_k | idle_r | sp | spa | v | data
error_pattern := on | off | single | pulse
error_size := 0- 28
symbol_select := bit bit bit bit
bit := 0 | 1
```

*Figure 1-12:* **BNF Description of the $bfm_change_param Call**

The parameters shown in Figure 1-12 are described in detail below:

- **error_type:** Takes the following values: scp, ecp, suf, snf, idle_a, idle_k, idle_r, sp, spa, v, cc, and data. Details of error_type are provided in "Symbol Corruption," page 34.
- **error_pattern:** Turns error injection on or off on a continuous basis, one shot or periodically, and can take the following values:

  **on:** Turn on error injection. Errors are inserted for all occurrences of error_type until turned off.

  **off:** Turn off error injection. Errors are not inserted for any occurrences of error_type until turned on.

  **single:** Apply error insertion to the next occurrence of error type only. Overrides *off* condition.

  **pulse:** Apply the parameters in the $bfm_pulse_event call to the same error_type.

- **error_size:** Describes the number of bits to corrupt and whether the control bit is to be corrupted or not. Takes the following values 0-8, 10-18, 20-28.

  **0-8:** Corrupt the defined number of bits. Locations are chosen randomly. Do not corrupt the control bit.

  **10-18:** Corrupt the defined number of bits (between 0-8). Locations are chosen randomly. Corrupt control bit.

  **20-28:** Delete 0-8 symbols of specified error_type.

  **Examples:**

  - 10 = Corrupt the control bit only.
  - 5 = Corrupt five bits of the byte, but do not corrupt the control bit.

- **symbol_select**: A 4-bit mask which turns on/off error injection of a symbol inside an ordered set. Note that for ordered sets which have two symbols, only the first two bits are used as symbol selects. For ordered sets which have one symbol, this parameter is ignored.

  Each bit in the 4-bit mask indicates whether error injection needs to be applied to the symbol. A 0 indicates no error injection and a 1 indicates error injection.

  **Default:** 1111. Indicates error injection needs to be applied to all symbols in an ordered set.

  **Example:** 1000. Indicates error injection is applied only to the first symbol of the ordered set.

## Symbol Corruption

The user can corrupt any symbol at any given instant in the simulation. The user does not have visibility into the internal state of the TX path of ABFM. The ABFM could be sending symbols from any of the following strings at a given instant: user PDU, UFC message, NFC message, idles, CCs. In addition, within a certain type of PDU/message, the ABFM could be sending the header (/SCP/, /SUF/, /SNF/), trailer (/ECP/) or data. Therefore, when a call is made to corrupt a certain type of symbol, it is applied to the next valid occurrence of the symbol.

If consecutive calls are made before the occurrence of a certain type of symbol, then only the most recent call is processed. For symbol corruption, the *error_size* parameter needs to be programmed to values between 0-18.

The following types of symbol corruption are provided:

- **/SCP/:** Corrupts one or both symbols in /SCP/. The symbols within /SCP/are chosen randomly at every call. The next User PDU is targeted for corruption.

  ```
  error_type = scp
  ```

- **/ECP/:** Corrupts one or both symbols in /ECP/. The symbols within /ECP/ are chosen randomly at every call. The next occurrence of /ECP/ is targeted for corruption.

  ```
  error_type = ecp
  ```

- **/PAD/:** Corrupts the next occurring /PAD/ symbol.

  ```
  error_type = pad
  ```

- **/SUF/**: Corrupts the /SUF/ symbol of the next UFC message.

  ```
  error_type = suf
  ```

- **/SNF/:** Corrupts the /SNF/ symbol of the next NFC message.

  ```
  error_type = snf
  ```

- **/A/,/K/,/R/:** Corrupts the next occurrence of the specified idle character.

  ```
  error_type = idle_a/idle_k/idle_r
  ```

- **/SP/:** Corrupts one or more symbols in the next occurrence of /SP/ sequence. The symbols corrupted are chosen randomly.

  ```
  error_type = sp
  ```

- **/SPA/:** Corrupts one or more symbols in the next occurrence of /SP/ sequence. The symbols corrupted are chosen randomly.

  ```
  error_type = spa
  ```

- **/V/:** Corrupts one or more symbols in the next occurrence of /V/ sequence. The symbols corrupted are chosen randomly.

  ```
  error_type = v
  ```

- **/CC/:** Corrupts one or more symbols in the next occurrence of the /CC/ sequence.

  ```
  error_type = cc
  ```

- **Data:** Corrupts the next data symbol.

  ```
  error_type = data
  ```

Examples of symbol corruption are shown in Figure 1-13.

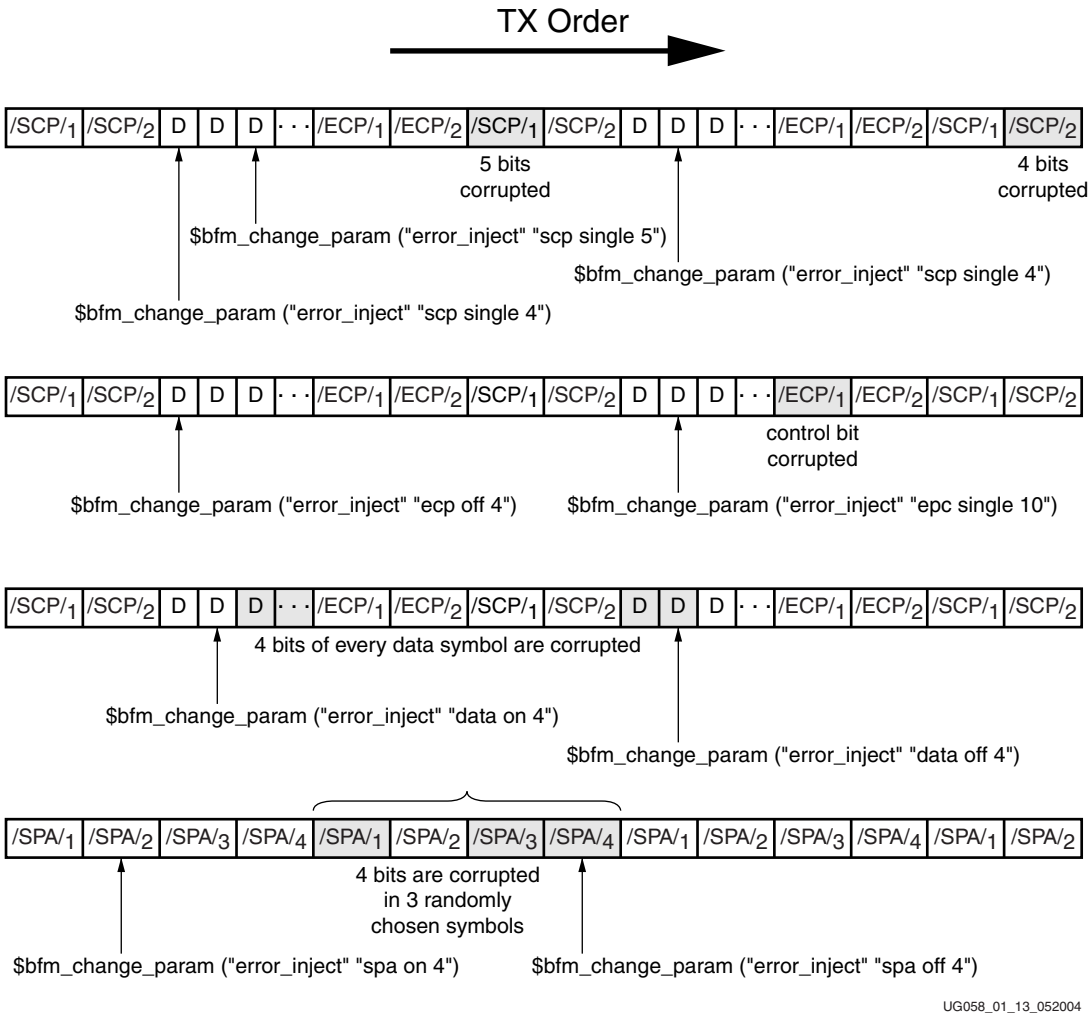


*Figure 1-13:* **Symbol Corruption Examples**

## Symbol Deletion

The user can delete any symbol at any given instant in the simulation. When a call is made to delete a certain type of symbol, it is applied to the next valid occurrence of the symbol. If consecutive calls are made before the occurrence of a certain type of symbol, then only the most recent call is processed. For symbol deletion, the *error_size* parameter needs to be programmed to values between 20-28. The second digit indicates the number of consecutive symbols to delete. The following types of symbol deletion are provided:

- **/SCP/:** Deletes one or both symbols in /SCP/. Valid error_size values are 20-22.

  ```
  error_type = scp
  ```

- **/ECP/:** Deletes one or both symbols in /ECP/. Valid error_size values are 20-22.

  ```
  error_type = ecp
  ```

- **/PAD/:** Deletes the next occurring /PAD/ symbol. error_size value is ignored.

  ```
  error_type = pad.
  ```

- **/SUF/:** Deletes the /SUF/ symbol of the next UFC message. error_size value is ignored.

  ```
  error_type = suf
  ```

- **/SNF/:** Deletes the /SNF/ symbol of the next NFC message. error_size value is ignored.

  ```
  error_type = snf
  ```

- **/A/,/K/,/R/:** Deletes one or more idle characters in the next occurrence of the specified idle character. Valid error_size values are 20-28.

  ```
  error_type = idle_a,idle_k,idle_r
  ```

- **/SP/:** Deletes one or more symbols in the next occurrence of /SP/ sequence. Valid error_size values are 20-24.

  ```
  error_type = sp
  ```

- **/SPA/:** Deletes one or more symbols in the next occurrence of /SPA/ sequence. Valid error_size values are 20-24.

  ```
  error_type = spa
  ```

- **/V/:** Deletes one or more symbols in the next occurrence of /V/ sequence. Valid error_size values are 20-24.

  ```
  error_type = v
  ```

- **/CC/:** Deletes one or more symbols in the next occurrence of the /CC/ sequence. Valid error_size values are 20-28.

  ```
  error_type = cc
  ```

- **Data:** Deletes one or more data symbols in the next occurrence of data symbols. Valid error_size values are 20-28.

  ```
  error_type = data
  ```

Examples of symbol deletion are shown in Figure 1-14.

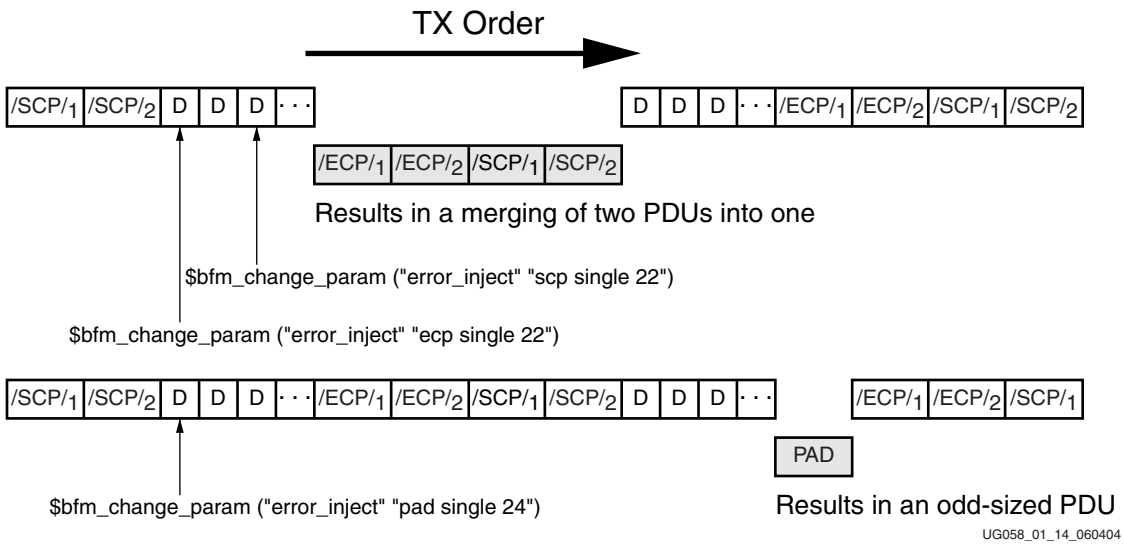


*Figure 1-14:* **Symbol Deletion Examples**

Periodic injection of errors can be accomplished using the $bfm_change_parameter call by setting error_pattern:= pulse (refer to "Error Injection," page 33). Whenever this setting is used, the $bfm_pulse_event call (refer to "$bfm_pulse_event(id,"sig name","width_str","freq_str","val_str",...);," page 49) can be used in conjunction with the $bfm_change_param call. Another requirement is that the *error_type* value in the $bfm_change_param call and the *sig_name* value in the $bfm_pulse_event call should be identical.

The $bfm_pulse_event call is described in BNF format as shown in Figure 1-15. A tabular format is also provided in Table 2-2, page 49.

```
$bfm_pulse_event(id, pulse_event_descriptor)
pulse_event_descriptor := "sig_name width_str freq_str val_str"
sig_name := error_type | src_wait_en
error_type := scp | ecp | suf | snf | idle_a | idle_k | idle_r | sp | spa | v | data
width_str := w inc min max
freq_str :=f inc min max
val_str := v pattern val
inc := integer
min := integer
max := integer
pattern := fixed | toggle | inc | rand
val := bit string (any number of bits)
```

*Figure 1-15:* **BNF Description of the $bfm_pulse_event Call**

Three examples of error injections are shown in Figure 1-16. In the first example, two calls are made. The first call, $bfm_change_param("error_inject", "scp pulse 4"), enables the pulse mode for the error injection and defines symbol corruption by corrupting four bits. The second call, $bfm_pulse_event ("scp" "w 1 1 1 0" "f 3 3 3 0","val fixed 1"), applies this error at a periodic rate. Errors are injected for one /SCP/ occurrence followed by two occurrences when there is no injection. This repeats every three occurrences. The remaining two examples show the variation of the error injection scenarios.

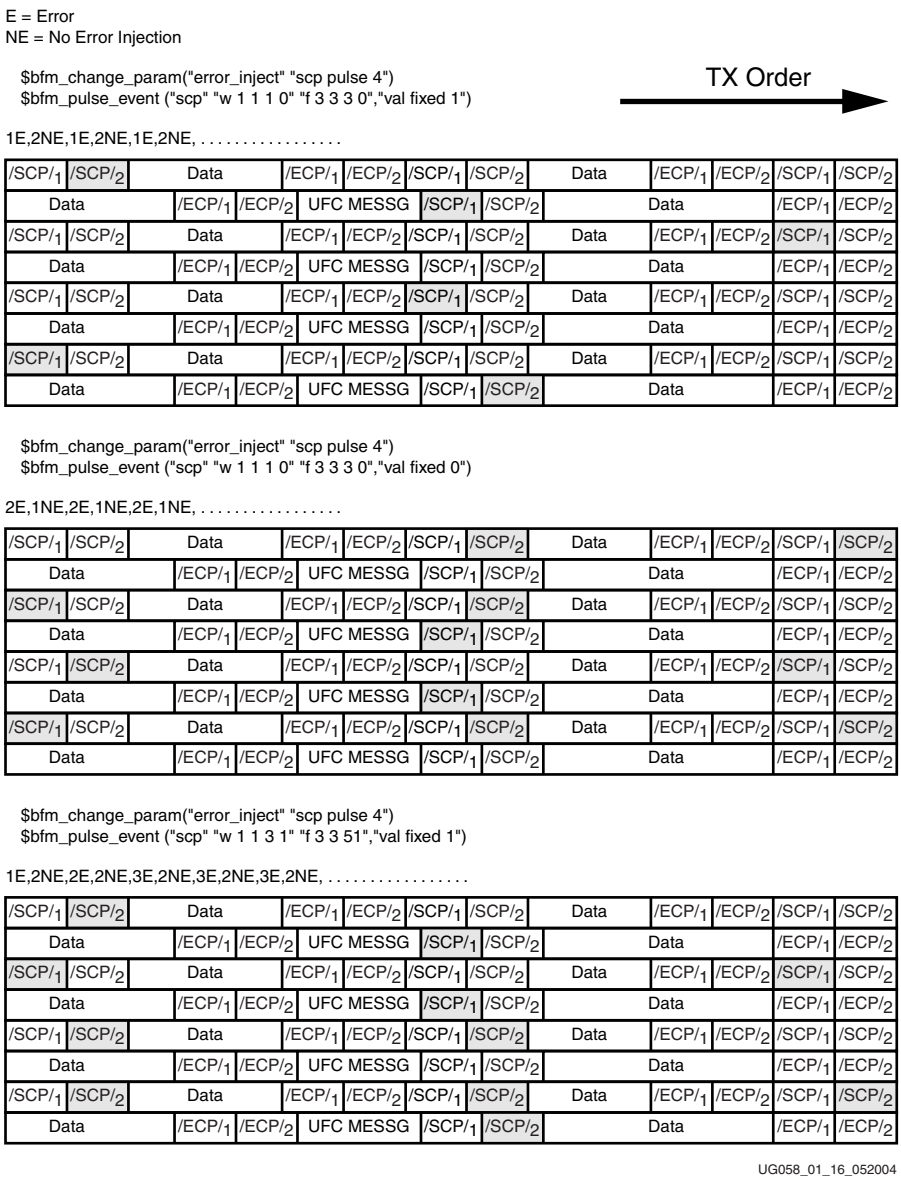


*Figure 1-16:* **Error Injection with Pulse Event**

XILINX®

Chapter 2

# PLI Interface and Test Bench Integration

## Scope

This chapter describes the details of the programming language interface (PLI) and its integration into the test bench. The PLI is a collection of functions written in C and is classified in two sections:

- "Library Functions"
- "User Functions"

## Library Functions

All simulators provide a PLI Library as part of the package. The PLI Library is based on the *IEEE 1364 Verilog PLI Standard*. It contains a large collection of predefined functions, which provide a powerful interface from C to Verilog. There are two categories of functions: TF and ACC. Both types of functions were used to develop the ABFM. These functions are similar to the system functions in C. Any further discussion on the Verilog PLI is beyond the scope of this document.

## User Functions

The user functions are written in C and use the library functions to build powerful routines to communicate with the Aurora architecture model (AAM) and the Verilog test bench. The user functions act as a master to the AAM and as a slave to the Verilog test bench. The user makes calls to these functions using system calls in the Verilog test bench. All system calls in Verilog start with the $ symbol. For example, $display and $finish are Verilog system calls that are built into the Verilog language. The ABFM system calls are prefixed with $bfm_ to differentiate them from the Verilog calls. The names of the user functions very closely match the names of the system calls. Because two different system calls can call the same user function, there cannot be an exact 1:1 match between system calls and user functions. A mapping structure describes the mapping between system calls and user functions, which is described in the *Verilog PLI Standard*.

***Note:*** The PLI is compiled into a shared library in binary format.

## Test Bench Integration

This section is the most relevant from the user's perspective. The user can make system calls in the test bench to communicate with the ABFM. It is recommended to instantiate the BFM calls in a separate Verilog wrapper whose input/output ports match that of the ABFM. The system calls provided in the ABFM are described in "System Calls," page 40.

## System Calls

All system calls support multiple instantiations of the ABFM. The very first argument to all system calls is *id*. To initialize two separate ABFMs, the following calls can be made:

- $bfm_initialize(1,…);
- $bfm_initialize(2,…);

### $bfm_initialize(id,in_port1,..,in_portn,out_port1,..,out_portm,arg1,..,argx)

This system call initializes the ABFM. The PLI directly probes the module (the Verilog wrapper) and identifies the input/output/inout ports and their widths based on the declarations in the module. This allows the ABFM to do a check to ensure that all input/output/inout ports are indeed passed as arguments to this system call. The system call expects all the ports and any additional arguments to be passed as arguments. This allows the PLI to store handles for all the arguments, which is mandatory. The first three arguments of this call are reserved for id, number of inputs, and number of outputs. The system call requires that the arguments occur in the following order:

1. Inputs (in the same order as declared in the module)
2. Outputs (in the same order as declared in the module)
3. Inouts (in the same order as declared in the module)
4. Other arguments (in an order chosen by the user)

While it is mandatory to declare all ports, additional arguments can be chosen based on necessity.

The Aurora tester module, `aurora_tester_pli.v`, has the following declarations for a four-lane single channel design:

```
input          clk;
input          rst;
output[0:3]    tx_p;
output[0:3]    tx_n;

input[0:3]    rx_p;
input[0:3]    rx_n;

reg  tx_fifo_full_ud, tx_fifo_full_uf, tx_fifo_full_nf;
reg  rx_fifo_full_ud, rx_fifo_full_uf, rx_fifo_full_nf;
reg  rx_fifo_empty_ud, rx_fifo_empty_uf, rx_fifo_empty_nf;
reg  rx_pkts_in_fifo_ud, rx_pkts_in_fifo_uf, rx_pkts_in_fifo_nf;
reg  [0:7]    conf_monitor;
reg  [0:12]   conf_result;
reg  [0:6]    init_monitor;
```

Therefore, the system call in the test bench is specified as follows:

```
initial
begin
$bfm_initialize(id,4,2,clk,rst,rx_p,rx_n,tx_p,tx_n,tx_fifo_full_ud,
tx_fifo_full_uf, tx_fifo_full_nf,rx_fifo_full_ud, rx_fifo_full_uf,
rx_fifo_full_nf,rx_fifo_empty_ud, rx_fifo_empty_uf,
rx_fifo_empty_nf,rx_pkts_in_fifo_ud, rx_pkts_in_fifo_uf,
rx_pkts_in_fifo_nf,conf_monitor,conf_result,init_monitor);
end
```

The above call is executed once at the start of the simulation. The FIFO full and empty signals are used to monitor the status of the internal memory of the ABFM. The user can monitor these signals to control sending packets to the TX FIFOs or draining packets from the RX FIFOs. The conf_monitor and conf_result signals are used to monitor the status of automatic protocol conformance. The signal init_monitor is used to monitor the status of the initialization of the ABFM. The exact position index of the arguments in this call needs to be maintained to map these signals to the internal structures in the ABFM. For example, the argument positions for conf_monitor, conf_result, and init_monitor should always be 22, 23, and 24.

Usage examples are provided under sections "$bfm_send_next_pkt_ud(id,no_of_pkts)," page 43 and "$bfm_get_next_pkt_ud(id,no_of_pkts)," page 44.

If additional ports and arguments are added to the module, then the AAM code and the system call arguments list change accordingly and alter the specification. The new ports/arguments add functionality to the ABFM. To track these values, additional handles need to be created in the AAM code. (If they do not affect the ABFM, they should not be declared in this module in the first place.)

## $bfm_execute(id)

This system call is the easiest to understand and to use. There are no arguments. All the handles are picked up by the $bfm_initialize call. Every call to this function does the following tasks in order:

1. Fetch all input values (from RX serial ports)
2. Execute RX path
3. Execute TX path
4. Drive all output values (to TX serial port outputs with respect to the ABFM)

The system call is used in the test bench as follows:

```
always @(posedge clk)
 begin
    $bfm_execute(1);
 end
```

The above usage calls the $bfm_execute every clock cycle. At every clock cycle, the state machines in the ABFM transition to the next state and update all the internal signals.

***Note:*** The clock used here is the serial clock for the RX path. This is used as the primary clock for the ABFM.

## $bfm_read_file_ud(id,"*filename*")

This system call opens file *"filename"* for reading and stores away the file pointer in a user_data file pointer. It expects *"filename"* to contain user data packets. A usage example is shown as follows:

```
initial
begin
$bfm_read_file_ud (1,"user_data_packets.dat");
end
```

The above function needs to be called only once.

### $bfm_read_file_uf(id,"*filename*")

This system call opens file *"filename"* for reading and stores away the file pointer in a user_flow file pointer. It expects *"filename"* to contain user flow packets. A usage example is shown as follows:

```
initial
begin
$bfm_read_file_uf (1,"user_flow_packets.dat");
end
```

The above function needs to be called only once.

### $bfm_read_file_nf(id,"*filename*")

This system call opens file *"filename"* for reading and stores away the file pointer in a native_flow file pointer. It expects *"filename"* to contain native flow packets. A usage example is shown as follows:

```
initial
begin
$bfm_read_file_nf (1,"native_flow_packets.dat");
end
```

The above function needs to be called only once.

### $bfm_write_file_ud(id,"*filename*")

This system call opens file *"filename"* for writing and stores away the file pointer in a user_data file pointer. The BFM dumps the packets received on the RX interface into this file. A usage example is shown as follows:

```
initial
begin
$bfm_write_file_ud (1,"rx_user_data_packets.dat");
end
```

The above function needs to be called only once.

### $bfm_write_file_uf(id,"*filename*")

This system call opens file *"filename"* for writing and stores away the file pointer in a user_flow file pointer. The BFM dumps user flow packets received on the RX interface into this file. A usage example is shown as follows:

```
initial
begin
$bfm_write_file_uf (1,"rx_user_flow_packets.dat");
end
```

The above function needs to be called only once.

### $bfm_write_file_nf(id,"*filename*")

This system call opens file *"filename"* for writing and stores away the file pointer in a native_flow file pointer. The BFM dumps native flow packets received on the RX interface into this file. A usage example is shown as follows:

```
initial
begin
$bfm_write_file_nf (1,"rx_native_flow_packets.dat");
end
```

The above function needs to be called only once

### $bfm_send_next_pkt_ud(id,no_of_pkts)

This system call picks the next *no_of_pkts* packets from the file opened using $bfm_read_file_ud call. The packet delineation is based on *start of file* (SOF) and *end of file* (EOF) tags. It stores the packets in a local, fixed-size array. To ensure that the call does not send more packets than the array can accommodate, it is recommended that the user poll the tx_fifo_full signal before sending more packets. An example usage is shown as follows:

```
initial
begin
#1000;
$bfm_send_next_pkt_ud(1,4);
#5000;
$bfm_send_next_pkt_ud(1,6);
end
```

This function can be called as many times as the user pleases. When *end of file* is reached, the BFM rewinds to *start of file* and continues fetching packets.

***Note:*** If the last byte of the fourth packet (in the above example) has not yet been transmitted when the next call is executed, though there is a gap of 5000 ns between the two calls, the BFM will treat it like a continuous stream of ten packets. The gaps between calls ensure there is no overflow of array memory, but do not always mean breaks in the data stream.

Another example of using the above system call to ensure that the ABFM sees a continuous stream of packets is as follows:

```
always @(posedge clk)
begin
if (tx_fifo_full_ud != 1)
   $bfm_send_next_pkt(1,1);
end
```

The above code ensures that as soon as there is room in the internal array to store another packet, the next packet is picked up by the BFM.

***Note:*** User flow messages can be embedded inside the user packets that are picked up with the user PDU.

### $bfm_send_next_pkt_uf(id,no_of_pkts)

The system call's description is the same as "$bfm_send_next_pkt_ud(id,no_of_pkts)," page 43 except that the packets in this file will be user flow packets.

This function can be used to send sideband user flow packets.

### $bfm_send_next_pkt_nf(id,no_of_pkts)

This system call can be used to send native mode messages. Although unlikely, this function can be used to send multiple packets back to back.

### $bfm_get_next_pkt_ud(id,no_of_pkts)

This system call drains the next *no_of_pkts* packets from the internal array and writes into a file opened using the $bfm_write_file_ud call. The packet delineation format used is based on SOF and EOF tags. An example usage is shown as follows:

```
initial
begin
#5000;
$bfm_get_next_pkt_ud(1,4);
#7000;
$bfm_get_next_pkt_ud(1,6);
end
```

This function can be called as many times as the user pleases.

The internal RX FIFOs store the data received from the RX distributor function in the ABFM. As the FIFOs are of fixed size, it can fill quickly if $bfm_get_next_pkt_* calls are not made frequently. The status of the FIFOs can be monitored from the test bench through signals rx_fifo_empty_* and rx_fifo_full_*. It is essential for the rx_fifo_empty_* and rx_fifo_full_* signals to be declared in the argument list of the $bfm_initialize call. This establishes a connection to the FIFO signals inside the ABFM.

The user should poll these bits and make calls to keep the FIFO in stable condition.

An example is shown below:

```
always @(posedge clk)
begin
if (rx_fifo_empty_ud == 0)
  if(rx_fifo_full_ud == 1)
    $bfm_get_next_pkt(1,1);
end
```

The above code ensures that as soon as the FIFO is full the next packet is drained by the ABFM. (Threshold values are set to ensure that when a *full* condition is reached, there is room for one more packet of max packet size = 1500 bytes). The FIFO conditions are discussed in "$bfm_change_param(id,param1,val,param2..,paramn,valn)," page 45, which addresses configuring the ABFM.

### $bfm_get_next_pkt_uf(id,no_of_pkts)

This system call drains packets from the user_flow array in the ABFM and written into a file. An option is provided in the ABFM to store embedded user flow messages in the user flow array. By default, embedded user flow messages are stored in user data array. It makes file comparisons easier.

### $bfm_get_next_pkt_nf(id,no_of_pkts)

This system call is used to log all the native mode messages received by the ABFM.

### $bfm_change_param(id,param1,val,param2..,paramn,valn)

This system call allows the user to modify parameters, configurations, and so forth in real time while the simulation is running. This must be used very judiciously. The arguments to this call must be provided in pairs, with the argument name first and the value second. The user can use this function to configure the number of lanes, the lane width, and whether in immediate or completion mode to suit a specific implementation of an Aurora design.

The names and possible values of arguments incorporated into the ABFM are listed in Table 2-1 (which spans multiple pages).

*Table 2-1:* **Arguments and Values for $bfm_change_param Call**

| Argument | Value(s) | Default Value | Description |
|---|---|---|---|
| cc_length | > 2 | 12 | Configures the length of a clock sequence per lane. |
| cc_frequency | > 2 | 10000 | Configures the frequency at which CC sequences need to be sent. |
| no_of_lanes | 1-32 | 4 | Configures the number of lanes. |
| imm_comp_mode | 0,1 | 0 | 0 = Immediate mode<br>1 = Completion mode |
| lane_skew | 0-100<br>format: "skew0 skew1..." | 0 | 0 = No skew<br>Indicates the transmit delay of serial bits in a lane. Lane skew can be achieved by setting different values for each lane. |
| src_wait_en | on, off, rand | rand | Injects idle cycles in PDU data to emulate source waits. |
| tx_fifo_thr_l | 1- 20,000 bytes | 1500 | Indicates the empty condition threshold for all FIFOs used in transmit path (user PDU, UFC, NFC). |
| tx_fifo_thr_h | 1- 20,000 bytes | 18500 | Indicates the full condition threshold for all FIFOs used in transmit path. |
| rx_fifo_thr_l | 1- 20,000 bytes | 1500 | Indicates empty condition threshold for all FIFOs used in receive path (user PDU, UFC, NFC). |

*Table 2-1:* **Arguments and Values for $bfm_change_param Call *(Continued)***

| Argument | Value(s) | Default Value | Description |
|---|---|---|---|
| rx_fifo_thr_h | 1- 20,000 bytes | 18500 | Indicates full condition threshold for all FIFOs used in receive path. |
| verbose | off, 1-5 | off | Indicates verbosity of log messages:<br>5 = Highest level of verbosity<br>1 = Lowest level of verbosity |
| error_inject | error_type | "data" | error_type can take the values:<br>scp, ecp, suf, snf, idle_a, idle_k, idle_r, sp, spa, v, cc, data |
| | error_pattern | off | error_pattern can take the values:<br>on, off, single, pulse |
| | error_size | 0 | error_size can take values:<br>0-18: For symbol corruption<br>20-28: For symbol deletion |
| | symbol_select | 1111 | Each bit of *symbol_select* can take values 0 or 1:<br>0 = No symbol corruption<br>1 = Symbol corruption |
| tx_polarity | 0,1 | 0 | 0 = Normal operation<br>1 = Invert serial tx polarity |
| rx_polarity | 0,1 | 0 | 0 = Normal operation<br>1 = Invert serial rx polarity |
| log_error | 0-3 | 0 | 0: Do not print error log in simulator's output<br>1: One shot: Print error log once<br>2: Print error log whenever a new error is encountered<br>3: Print error log for the next error only |
| chk_init | 0,1 | 0 | 0 = Normal mode<br>1 = Enable initialization conformance |
| chk_cup_conf | 0,1 | 0 | 0 = Normal mode<br>1 = Enable channel up conformance |
| chk_softerr_conf | 0,1 | 0 | 0 = Normal mode<br>1 = Enable soft error conformance |
| chk_conf | 0,1 | 0 | 0 = Normal mode<br>1 = Enable all phases of conformance checks<br>Overrides chk_init, chk_cup_conf and chk_softerr_conf settings. |

*Table 2-1:* **Arguments and Values for $bfm_change_param Call *(Continued)***

| Argument | Value(s) | Default Value | Description |
|---|---|---|---|
| same_databeat_pkts | 0,1 | 1 | 0 = Only one packet of same kind can be transmitted in same data beat<br>1 = Allow more than one packet of same kind (user, UFC, NFC) to be transmitted in same data beat |
| pdu_lane_start | seq, rand, 0,…, no_of_lanes-1 | 0 | seq = Starting from lane 0, every successive PDU is sent on the next lane in a round robin fashion<br>rand = Lane number is picked randomly<br>0,..., no_of_lanes -1 = PDU is sent on the indicated lane |
| echo mode | 0,1 | 1 | 0 = Normal mode<br>1 = Sets BFM in *echo* mode. RX looped back to TX |
| sym_conf_len | 0-4 | 4 | Indicates the number of symbols used to match two ordered sets during protocol conformance. For example, 2 indicates that the receive side will only check the first two symbols of /SP/,/SPA/ or /V/ to qualify an *ordered set match* |
| system_latency | any integer > 0 | 20 | Worst latency of the DUT (from link interface to serial interface) in units of Aurora cycles.<br>**Note:** A conservative number is recommended while modifying this parameter.Used during protocol conformance. |
| first_sp_thr | any integer > 0 | 200 | Minimum number of Aurora clock cycles after which the DUT is guaranteed to send an /SP/.<br>**Note:** A conservative number is recommended while modifying this parameter.Used during protocol conformance. |
| echo_fifo_depth | any integer > 0 | 40 | Depth of the external echo FIFO which connects the DUT's RX to TX.<br>**Note:** A conservative number is recommended while modifying this parameter. Used during protocol conformance. |

An example usage is:

```
initial
begin
$bfm_change_param (1,no_of_lanes,4,codes_per_lane,2,imm_mode,0);/*804 implementation, completion
mode*/
$bfm_initialize (1,...,no_of_lanes,codes_per_lane,tx_fifo_thr_h,rx_fifo_thr_l,rx_fifo_thr_h);
end
initial
begin
$bfm_change_param (1,no_of_lanes,4,codes_per_lane,2);/*804 implementation*/
#5000
$bfm_change_param (1,tx_fifo_thr_h,19000);
#7000;
$bfm_change_param (1,rx_fifo_thr_l,1000,rx_fifo_thr_h,18000);
end
```

The above code can be used to configure the full and empty thresholds for the TX FIFOs and RX FIFOs. This provides a mechanism to modify the internal parameters of the ABFM in a convenient way.

### $bfm_reset(id)

This system call allows the user to reset the ABFM. This call transfers control of the ABFM to the internal initialization routine. During internal initialization, the FIFOs are flushed, the state machines are reset, and all parameters are set to their initial values. The user can choose to call this function together with the rst signal for the design.

### $bfm_get_simulation_time(id)

This system call allows the user to get the simulation time at any given point in the simulation run.

### $bfm_pulse_event(id,"sig name","width_str","freq_str","val_str",...);

This call is used to create a periodic or random event on any signal or data bus. The syntax for this call is as follows:

```
$bfm_pulse_event (1,"sig name 1","width_str", "freq_str", "val_str",
"sig name 2",...);
```

Table 2-2 shows the four parameters for the pulse event call.

*Table 2-2:* **Pulse Event Parameters**

| Parameter | String | Description |
|---|---|---|
| sig name | | Can be a signal name or an internal variable in the ABFM |
| width_str | w | Qualifies the current parameter as a width parameter |
| | inc | Indicates the increase in pulse width at the next pulse event |
| | min | Starting pulse width of the signal |
| | max | Ending pulse width of the signal<br>**Note:** No more increments of width are performed if the increment results in a value larger than max. |
| freq_str | f | Qualifies the current parameter as a frequency (period) parameter |
| | inc | Indicates the increase in period at the next pulse event |
| | min | Starting period of the signal |
| | max | Ending period of the signal<br>**Note:** No more increments of period are performed if the increment results in a value larger than max. |
| val_str | v | Qualifies the current parameter as being the value of the signal pulse |
| | fixed/toggle/ inc/rand | At every period of the signal, the new value of the signal pulse can be one of the following:<br>• fixed - the value determined by the *val* entry<br>• toggle - toggle the value of the previous signal pulse<br>• inc - increment the value<br>• rand - any arbitrary value<br>**Note:** For 1-bit signals, toggle and inc achieve the same result. |
| | val | The value of the signal pulse when the *fixed* type is used<br>**Note:** *val* fixes the value of the signal during the pulse width phase. During the rest of the period, the value will be the complement. |

Examples of the pulse event call and associated parameters are:

- $bfm_pulse_event (1,"clk_div","w 0 1 1","f 0 2 2","v fixed 0") generates a clock signal that is half the frequency of the ABFM clock.
- $bfm_pulse_event (1,"data", "w 1 1 1", "f 2 2 2", "v inc 1") (where data is an 8-bit bus) creates the hexadecimal data 1h, feh, 2h, 2h, fdh, fdh, 3h, 3h, 3h, fch, fch, fch, …

## Packet Generator

A packet generator script is used to generate stimulus files used by the test bench. It uses a simple formatting technique that delineates packets using #SOF and #EOF tags. Packets can be configured with the options shown in Table 2-3.

*Table 2-3:* **Packet Configuration Options**

| Option | Default Value | Description |
|---|---|---|
| -bw | 32 | A bus width option that takes integer values as input |
| -p | 10 | Determines the number of packets |
| -l | 64 bytes | Packet length<br>If the value is an integer, then the packet length is fixed to that integer.<br>If the value is *var*, then the packet length is variable and is randomly picked between 1 and the max_pkt_length value as determined by the -maxl option. |
| -maxl | 1,500 bytes | Maximum packet length (max_pkt_length)<br>This takes integer values and fixes an upper limit on the size of the packet. This is relevant only when *-l var* option is used. |
| -ty | ud | Packet type: user data or user flow<br>ud = User data packet<br>uf = User flow packet (not yet implemented) |
| -seq or -rand | -seq | Determines data byte order within a packet<br>-seq = Sequential<br>-rand = Random numbers between 0 and 255 |

The user can type **`packet_gen.pl -h`** to display the packet generator options such as:

```
-bw <bus_width>, in bytes. default is 32
-p <no_of_pkts>, default is 10
-l <pkt_len>, takes integer values (default is 100 )
     or "var"(randomized packet lengths)
-maxl <max_pkt_len>, takes integer values (default is 1500)
-ty <packet_type>,"ud"(user_data) or "uf"(user flow)
     default is user data
-seq or -rand, data values are generated sequentially or random
              default is sequential
-o <output_file>, name of output file
     default names for "ud" and "uf" are user_data_packets.dat
     and user_flow_packets.dat
```

## Simulation Procedure

Simulations can be run using the ABFM with minimal changes to the test environment. ABFM simulations involve the following steps:

1. Instantiate a BFM module

   To instantiate a BFM module, first create a module that contains all of the BFM calls and ports that interface to the rest of the test bench environment. The `aurora_tester_pli.v` file described in the "System Calls," page 40, can be used as a starting point.

2. Generate packet files

   To generate packet files, use the packet generator script described in "Packet Generator," page 50. The packet generator command is added under the **runsim** target in **makefile**. This causes a new packet data file to be created whenever a simulation is run.

3. Run simulations

   To run simulations with a BFM, an extra option must be declared at run time. For the ModelTech environment, **`-pli pli_shared_object.sl`** needs to be added to the **`runtime`** command. The current implementation uses the following command:

   ```
   "vsim -c -do vsim.do testbench.v  -pli abfm.sl "
   ```

4. Check results

   To check results, a simple **checksim** routine strips comment lines from the input file and output file and compares them using a *diff* function. It echoes a `SIMULATION PASSED` or `SIMULATION FAILED` message based on the result.

   All of the above steps are incorporated into **makefile**, making it relatively easy to use.

XILINX®

Chapter 3

# FLI Interface

## Scope

The foreign language interface (FLI) provides the ability to interface with the BFM shared library from a VHDL environment. It can be considered as a VHDL equivalent of PLI. FLI is a proprietary language which was developed to provide an interface between VHDL and C/C++ routines only in a ModelSim simulator environment. The FLI and PLI share the Aurora architecture model (AAM). The user functions written for FLI use proprietary structures provided by the ModelSim simulator to communicate with a VHDL test bench.

There are three components required to run a simulation that interfaces with the BFM:

- BFM shared library: `abfm.sl`
- Foreign subprograms file: `abfm_fli_subprograms.vhd`
- Initialization function

## Simulation Components

### BFM Shared Library

The BFM shared library `abfm.sl` is included in the release. This library is compiled using the AAM and user functions written specifically for the FLI.

### Foreign Subprograms File

The foreign subprogram file `abfm_fli_subprograms.vhd` is included with the release. This file contains the procedure definitions for all the foreign subprograms (equivalent to system calls in the PLI) that are called from the VHDL test bench. This file must be compiled at simulation time.

A foreign subprogram example is shown below for bfm_change_param, the equivalent of a $bfm_change_param (1,"no_of_lanes","4") in a PLI environment. The number of arguments defined in the user function must explicitly map to the number of arguments in the foreign subprogram call. This affects only the bfm_change_param subprogram.

Foreign subprogram definition in `abfm_fli_subprograms.vhd`:

```
procedure bfm_change_param(
  id : IN integer;
  param_name   : IN string;
  param_val1   : IN string;
  param_val2   : IN string
  );
  attribute foreign of bfm_change_param : procedure is "bfm_change_param abfm.sl";
procedure bfm_change_param(
      id : IN integer;
      param_name   : IN string;
      param_val1   : IN string;
      param_val2   : IN string
    ) is
    begin
      assert false report "ERROR: foreign subprogram not called" severity note;
    end;
```

Foreign subprogram call in the test bench:

```
init_cfg: process
  begin
    bfm_change_param(id,"no_of_lanes","4","");
    wait
  end process;
```

Although the fourth parameter is not relevant, this empty parameter is needed. The PLI has the ability to scan the arguments and append *empty* arguments to satisfy the user function requirements. The FLI, however, cannot do the same, therefore, the empty argument is required. This applies only to the bfm_change_param subprogram. Sample test benches provided in the release further illustrate the usage of the foreign subprograms.

## Initialization Function

The initialization function is called and executed during the elaboration phase of the simulator. Only the bfm_initialize function is defined and executed during this phase. It needs to be defined as a separate entity in the test bench. The architecture for this entity must be defined as follows:

```
ENTITY bfm_initialize IS
   port (
         id :  in integer :=0;
         clk  :  in bit := '0';
         rst  :  in bit := '0';
         rx_p  :  in bit_vector(3 downto 0) := "1010";
         rx_n  :  in bit_vector(3 downto 0) := "1010";
         tx_p  :  out bit_vector(3 downto 0) ;
         tx_n  :  out bit_vector(3 downto 0) ;
         tx_fifo_full_ud: out bit;
         tx_fifo_full_uf: out bit;
         tx_fifo_full_nf: out bit;
         rx_fifo_full_ud: out bit;
         rx_fifo_full_uf: out bit;
         rx_fifo_full_nf: out bit;
         rx_fifo_empty_ud: out bit;
         rx_fifo_empty_uf: out bit;
         rx_fifo_empty_nf: out bit;
         rx_pkts_in_fifo_ud: out integer;
         rx_pkts_in_fifo_uf: out integer;
         rx_pkts_in_fifo_nf: out integer
         );
END bfm_initialize;

ARCHITECTURE c_model OF bfm_initialize IS
  attribute foreign : string;
  attribute foreign of c_model : architecture is "bfm_initialize ./abfm.sl";
  begin
  end;
```

Sample test benches provided in the release further illustrate the usage of the initialization function.

Although the system calls and functions in PLI begin with $, the foreign subprograms and initialization calls in FLI cannot begin with $.

XILINX®

# Appendix A

# *Simplex with a Serial Interface*

## Introduction

The original architecture model is targeted for duplex designs. This appendix extends the architecture to include simplex designs.

The Aurora simplex BFM supports the verification of Aurora simplex designs. A single BFM is provided to be configured to work as a transmit (TX) BFM, a receive (RX) BFM, or both a TX and RX BFM.

The BFM wrapper files are automatically generated along with Aurora reference designs through the CORE Generator™ tool configured in the chosen mode of simplex operation.

The simplex BFM inherits all features in the duplex BFM, with the exception of the automatic protocol compliance.

## Architecture

The Aurora simplex BFM architecture is shown in Figure A-1, page 58. This architecture is derived by partitioning the duplex architecture into TX/RX and further replacing internal handshake between TX and RX during initialization with external sideband ports. The Aurora simplex BFM can be used in three scenarios:

- **Scenario 1:** A simplex TX BFM talking to a simplex RX DUT (Figure A-2, page 59)
- **Scenario 2:** A a simplex RX BFM talking to a simplex TX DUT (Figure A-3, page 59)
- **Scenario 3:** A a single TX and RX BFM in "both" mode talking to one simplex TX DUT and one simplex RX DUT (Figure A-4, page 60)

***Note:*** For details on the Aurora duplex BFM operation, see the section "Aurora Architecture Model," page 17.)

## Aurora Simplex BFM Architecture (TX and RX)

Figure A-1 shows the Aurora simplex BFM architecture (TX and RX).

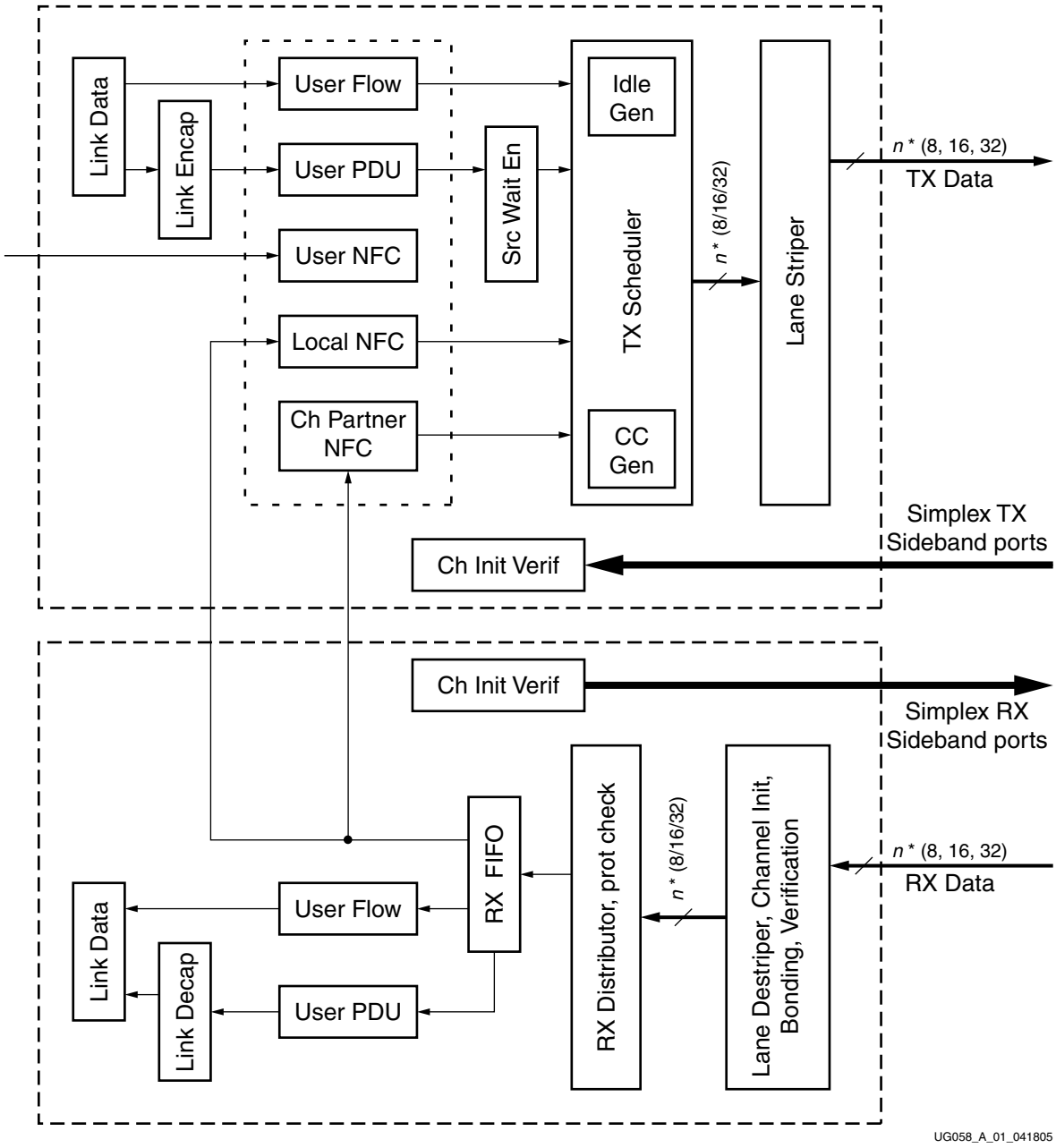


*Figure A-1:* **Aurora Simplex BFM Architecture (TX and RX)**

## Aurora Simplex TX BFM

Figure A-2 shows the Aurora simplex TX BFM module in the test environment with a simplex RX DUT.

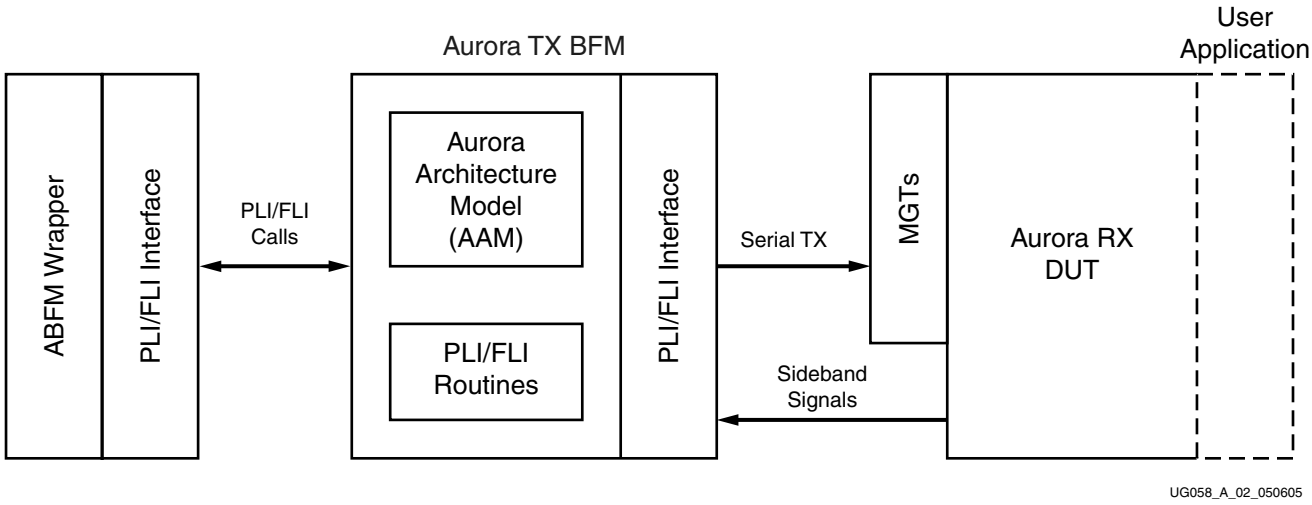


*Figure A-2:* **Aurora Simplex TX BFM: Test Environment with Simplex RX DUT**

## Aurora Simplex RX BFM

Figure A-3 shows the Aurora simplex RX BFM module in the test environment with a simplex TX DUT.

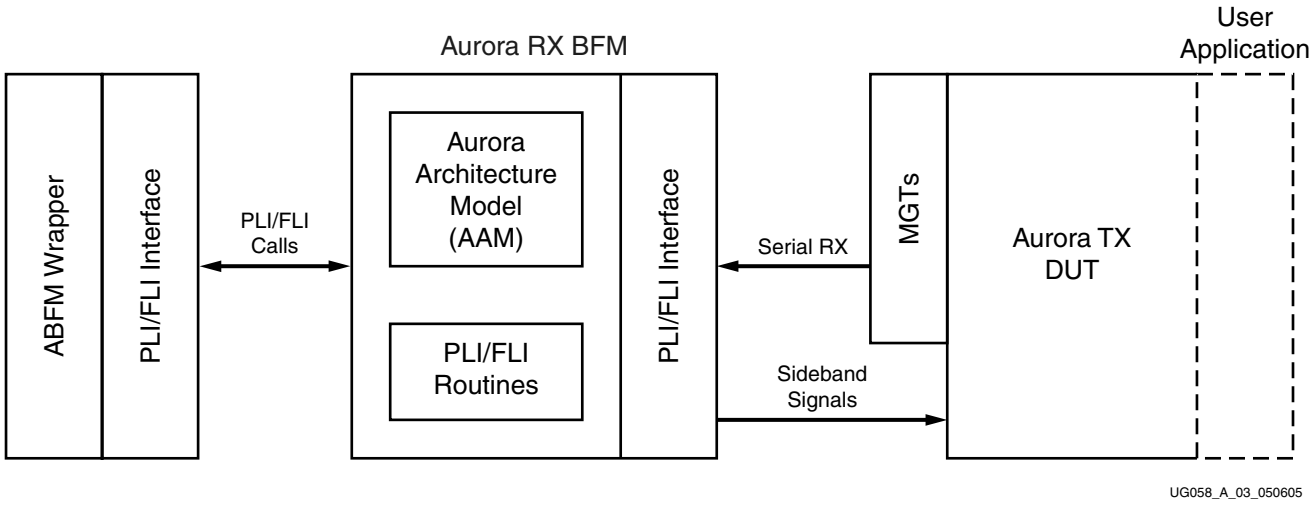


*Figure A-3:* **Aurora Simplex RX BFM: Test Environment with Simplex TX DUT**

## Aurora Simplex "Both" BFM

Figure A-4 shows the Aurora simplex "Both" BFM module in the test environment with a simplex RX DUT and a simplex TX DUT.

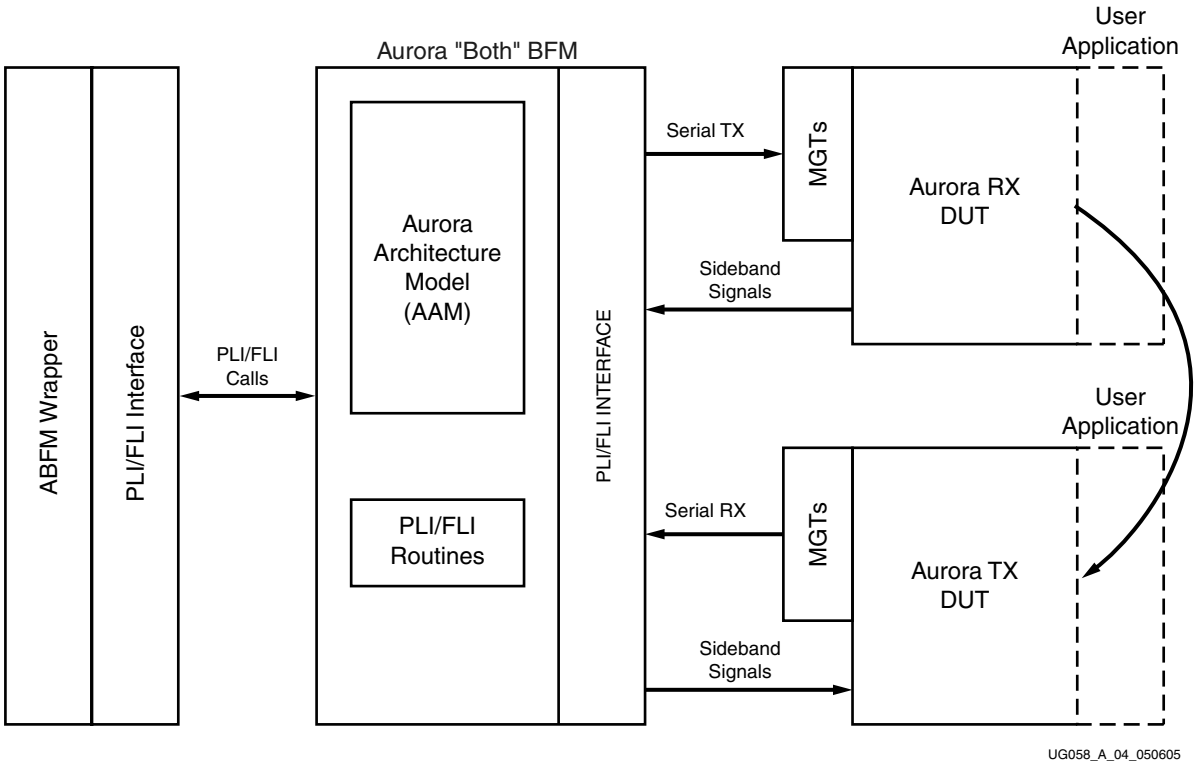


*Figure A-4:* **Aurora Simplex "Both" BFM: Test Environment with Simplex RX and TX DUTs**

The additional sideband signals for the simplex mode BFMs are described in Table A-1 and Table A-2.

*Table A-1:* **Sideband Signals for Simplex TX**

| Label | Direction | Description |
|---|---|---|
| TX_ALIGNED | Input | Connects to RX_ALIGNED signal of partner. |
| TX_BONDED | Input | Connects to RX_BONDED signal of partner. |
| TX_VERIFIED | Input | Connects to RX_VERIFIED signal of partner. |
| TX_RESET | Input | Connects to RX_RESET signal of partner. |

*Table A-2:* **Sideband Signals for Simplex RX**

| Label | Direction | Description |
|---|---|---|
| RX_ALIGNED | Output | Connects to TX_ALIGNED signal of partner. |
| RX_BONDED | Output | Connects to TX_BONDED signal of partner. |
| RX_VERIFIED | Output | Connects to TX_VERIFIED signal of partner. |
| RX_RESET | Output | Connects to TX_RESET signal of partner. |

# PLI Interface and Test Bench Integration

The PLI interface for the simplex BFM is very similar to the duplex BFM with a few modifications. The modifications affect only the system call $bfm_initialize. The port descriptions and parameters are defined in "$bfm_initialize" and "$bfm_change_param," page 62.

## $bfm_initialize

```
$bfm_initialize(id,mode,no_of_inputs,no_of_outputs,in_port1,..,in_port
n,out_port1,..,out_portm,arg1,..argx);
```

The *mode* port in the above system call can take the following values:

- 1 := Simplex TX
- 2 := Simplex RX
- 3 := Both (Simplex TX and RX)

The following statements enable accurate port mapping between the ABFM and the test bench. The BFM package includes examples that shows how the wrapper for the BFM must be written. A specific 804 simplex TX BFM example is shown below:

```
input  clk;
input  rst;
output [0:3] tx_p;
output [0:3] tx_n;
input  tx_aligned;
input  tx_bonded;
input  tx_verified;
input  tx_reset;
reg    tx_fifo_full_ud,tx_fifo_full_uf,tx_fifo_full_nf;
reg    tx_fifo_full_ud,rx_fifo_full_uf,rx_fifo_full_nf;
reg    rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf;

reg    [0:7] conf_monitor;
reg    [0:12] conf_result;
reg    [0:6] init_monitor;
```

The system call in simplex TX is:

```
$bfm_initialize(id,1,6,2,clk,rst,tx_aligned,tx_bonded,tx_verified,tx_r
eset,tx_p,tx_n
tx_fifo_full_ud,tx_fifo_full_uf,tx_fifo_full_nf,
rx_fifo_full_ud,rx_fifo_full_uf,rx_fifo_full_nf,
rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf,
conf_monitor,conf_result,init_monitor);
```

## $bfm_change_param

```
$bfm_change_param(id,param1,val,..,paramn,val)
```

The parameters shown in Table A-3 are added to the BFM parameter list to configure the behavior of the simplex BFM.

The system call usage and the majority of the parameter definitions are the same as described in "System Calls," page 40.

*Table A-3:* **Additional Parameters for Simplex BFM Operation**

| Argument | Value(s) | Default Value | Description |
|---|---|---|---|
| simplex_tx_auto_init | 000 to 111 | 000 | The three bits correspond to auto completion of each phase of initialization: *Alignment*, *channel bonding*, and *channel verification*.<br>• 0 : No auto completion<br>• 1 : Auto completion<br>The state machine will exit the current phase indicating success after a fixed number of symbols is received.<br>This is used in conjunction with parameters simplex_tx_align_timer, simplex_tx_bond_timer, and simplex_tx_verify_timer |
| simplex_tx_align_timer | Any integer value | 100 | Indicates the number of symbols after which the state machine will exit the phase and set tx_aligned to 1. |
| simplex_tx_bond_timer | Any integer value | 200 | Indicates the number of symbols after which the state machine will exit the phase and set tx_bonded to 1. |
| simplex_tx_verify_timer | Any integer value | 300 | Indicates the number of symbols after which the state machine will exit the phase and set tx_verified to 1. |
| rx_reset_pulse_width | Any integer value | 1 | Indicates the number of Aurora cycles for which the rx_reset signal will remain at 1 when hard errors are encountered causing BFM to reset. |

Appendix B

# Aurora BFM: Parallel Interface

## Introduction

The Aurora BFM (ABFM) incorporates the functionalities of Aurora with those of the multi-gigabit transceiver (MGT) to comprehensively test the DUT together with the MGT behavior. The serial behavior of the MGT, however, can greatly increase the simulation time.

The user can bypass the MGT functionality, though, by using a parallel interface to connect the Aurora DUT directly to the BFM. A parallel interface provides the following benefits:

- Increased simulation speed as the base unit of the simulation is not a serial clock cycle. It can run 10x/20x/40x faster depending on the interface width of the serial device.
- Users can implement their own SERDES/transceiver technologies (which might or might not be compatible with the MGT modeling in the ABFM) can still simulate their Aurora designs with the ABFM parallel interface.
- Users can test the DUT with MGT functionality by connecting the parallel interface to an MGT swift model in the test bench.

# Architecture

The architecture of the parallel interface is shown in Figure B-1.

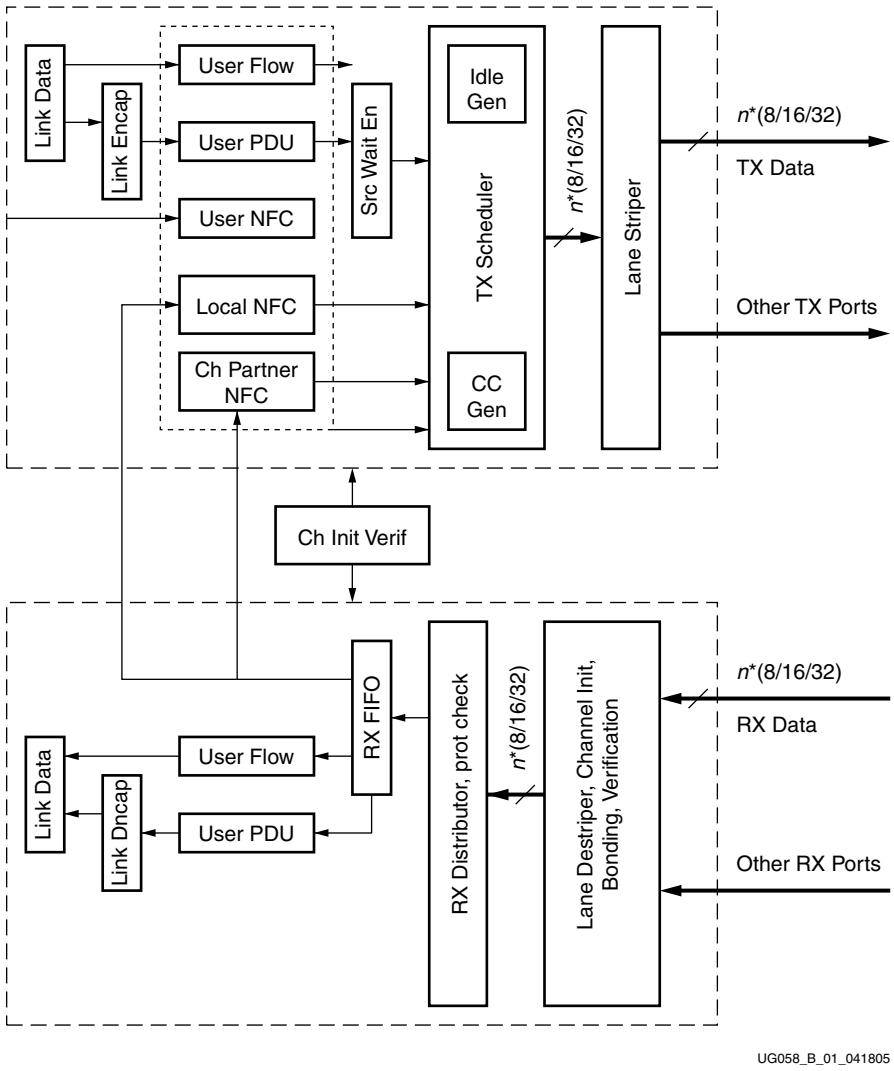


*Figure B-1:* **ABFM Parallel Interface Architecture**

This architecture is derived by removing the MGT functionality from the ABFM with serial interface (see "Aurora Architecture Model" in Chapter 1 for details on the ABFM with serial interface). The parallel interface ABFM can be used in the two scenarios shown in Figure B-2 (Scenario 1) and Figure B-3 (Scenario 2):

- **Scenario 1:** Shows how a user with an existing environment consisting of the ABFM with serial interface can switch to a parallel interface by instantiating the MGT swift models (on either side)
- **Scenario 2:** Shows a direct connection between the DUT and the ABFM which is enabled by bypassing the MGTs. This significantly speeds up simulation times

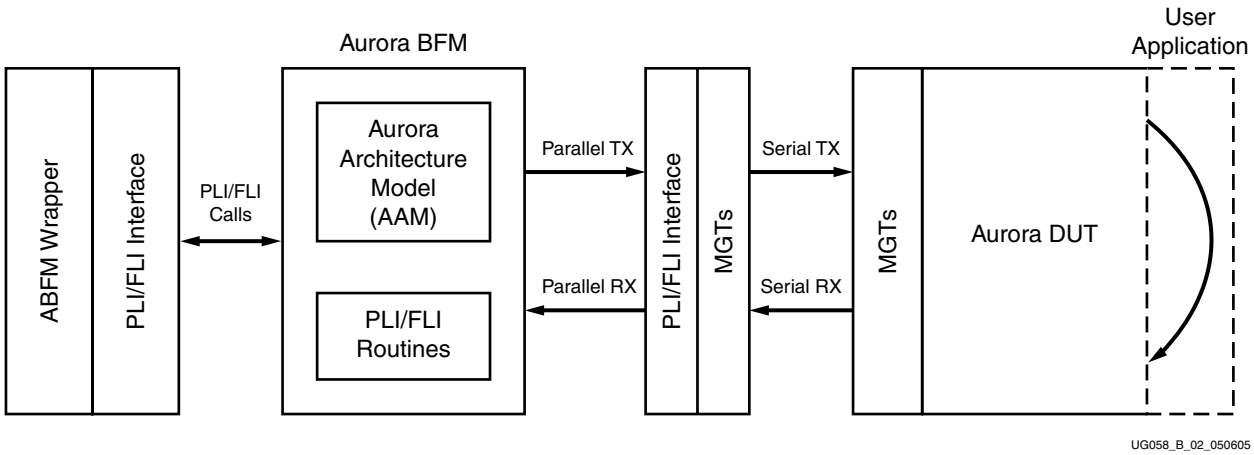


*Figure B-2:* **BFM Parallel Interface: Test Environment with MGT Swift Models**

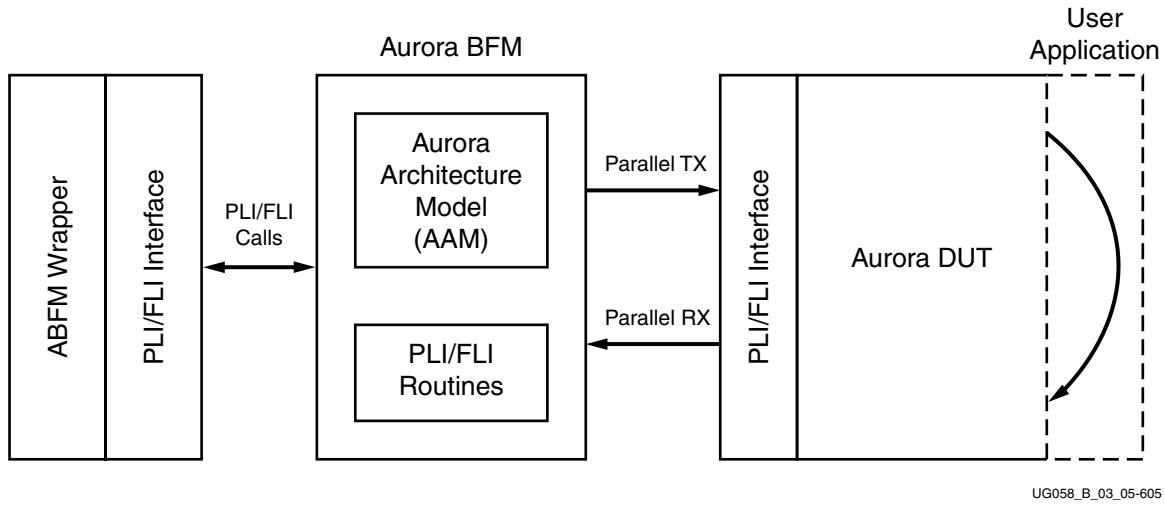


*Figure B-3:* **ABFM Parallel Interface: Test Environment with No MGTs**

# Signals and Parameters

The width of the signals in the parallel interface scales according to the configuration of the Aurora DUT. The list of signals defined at the parallel interface are described below.

## Parameters

WBYTES = Lane width in bytes

WBITS = Lane width in bits

N = Number of lanes

## Generic Signals

```
CLK
  /* Clock input from testbench */
RST
  /* Reset input from testbench */
```

## Transmit Path Signals

```
TX_DATA[WBITS*N - 1:0]
  /*OUTPUT*/
  /*scales with MGT lane width and No of Lanes*/
  /*Connects to RX_DATA of partner or TX_DATA of MGT*/

TX_CHAR_IS_K_EM[WBYTES*N - 1:0]
  /*OUTPUT*/
  /*scales with MGT Lane width and No of Lanes*/
  /*Connects to RX_CHAR_IS_K of partner or TX_CHAR_IS_K of MGT*/

TX_CHAR_IS_COMMA_EM[WBYTES*N - 1]
  /*OUTPUT*/
  /*scales with MGT lane width and No of Lanes*/
/*Valid only in a non-MGT environment. Emulates the behavior of the MGT
  generated CHAR_IS_COMMA on the RX side */

// Comma Detect Signals
ENA_COMMA_ALIGN_EM [N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes */
  /*connects to ENA_COMMA_ALIGN input of MGT*/
  /*valid only in an MGT environment. Leave unconnected otherwise*/

ENMCOMMAALIGN_EM[N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes*/
  /*connects to ENMCOMMALIGN input of MGT*/
  /*valid only in an MGT environment. Leave unconnected otherwise*/

ENPCOMMAALIGN_EM[N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes*/
  /*connects to ENPCOMMAALIGN input of MGT */
  /*valid only in an MGT environment. Leave unconnected otherwise */
```

```
ENCHANSYNC_EM[N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes */
  /*valid only in a multi-lane MGT environment */

CHBONDDONE_OUT_EM[N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes */
  /*used only in a non-MGT environment */
  /*connects to CHBONDDONE signal of Partner. It emulates
   the MGT CHBONDONE out signal */

TX_BUF_ERR[N-1:0]
  /*INPUT*/
  /*scales with No of Lanes */
  /*Connects to TX_BUF_ERR output of MGT */
  /*valid only in an MGT environment. Leave unconnected otherwise*/

TX_K_ERR [WBYTES*N - 1:0]
  /*INPUT*/
  /*scales with MGT lane width and No of Lanes*/
  /*Connects to TX_K_ERR output of MGT */
  /*valid only in an MGT environment. Leave unconnected otherwise*/

TX_NOT_IN_TABLE_EM[WBYTES*N - 1]
  /*OUTPUT*/
  /*scales with MGT lane width and No of Lanes*/
  /* Emulates the behavior of the MGT generated NOT_IN_TABLE on RX side*/
  /* connects to RX_NOT_IN_TABLE of partner */
  /* Used only in a non-MGT environment. Leave unconnected otherwise*/

TX_DISP_ERR_EM[WBYTES*N - 1]
  /*OUTPUT*/
  /*scales with MGT lane width and No of Lanes*/
  /*Emulates the behavior of the MGT generated DISP_ERR on the RX side */
  /* connects to RX_DISP_ERR of partner */
  /*Used only in a non-MGT environment. Leave unconnected otherwise*/

TX_RESET_EM[N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes */
  /* used to reset the MGT in an MGT environment. Leave unconnected
  otherwise*/

RX_POLARITY_IN[N-1:0]
  /*INPUT*/
  /*scales with No of Lanes */
  /*valid only in non-MGT environment. This input is tied to the
    RX_POLARITY Output from partner. BFM responds by switching the
    polarity of it's TX*/
  /* when MGT is present, tie to '0' in the testbench*/
```

## Receive Path Signals

```
RX_DATA [WBITS*N - 1:0]
  /*INPUT*/
  /* scales with MGT lane width & No of Lanes*/
  /* connects to TX_DATA of partner or RX_DATA of MGT*/

RX_CHAR_IS_K[WBYTES*N - 1:0]
  /*INPUT*/
  /*scales with MGT lane width and No of Lanes*/
  /*connects to TX_CHAR_IS_K of partner or RX_CHAR_IS_K of MGT*/

RX_CHAR_IS_COMMA[WBYTES*N - 1:0]
  /*INPUT*/
  /*scales with MGT lane width and No of Lanes*/
  /*valid only in an MGT environment */

RX_NOT_IN_TABLE[WBYTES*N -1:0]
  /*INPUT*/
  /*scales with MGT lane width & No of Lanes*/
  /*valid only in an MGT environment.
  Tie off to "0" otherwise*/

RX_DISP_ERR[WBYTES*N - 1:0]
  /*INPUT*/
  /*scales with MGT lane width and No of Lanes*/
  /*valid only in an MGT environment. Tie off to "0" otherwise*/

RX_BUF_STATUS[N-1:0]
  /*INPUT*/
  /*scales with No of Lanes */
  /*valid only in an MGT environment.
  Tie off to '0' otherwise */

RX_CLK_COR_CNT[3*N-1:0]
  /*INPUT*/
  /*scales with No of Lanes */
  /*connects to rx_clk_cor_cnt signal of MGT*/
  /*valid only in an MGT environment.
  Tie off to '0' otherwise */

CHBONDDONE[N-1:0]
  /*INPUT*/
  /*scales with No of Lanes */
  /*connects to CHBONDDONE signal of MGT*/
  /*valid only in an MGT environment. Tie to '1'otherwise */

RX_POLARITY[N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes */
  /*valid only in an MGT environment. This output is tied to the
  /* RX_POLARITY input of MGT */

RX_RESET_EM [N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes */
  /*connects to RX_RESET of MGT. Used to reset the MGT */
  /*valid only in an MGT environment */
```

```
RX_REALIGN_EM[N-1:0]
  /*OUTPUT*/
  /*scales with No of Lanes */
  /*connects to RX_REALIGN of MGT. Used to realign the MGT */
  /*valid only in an MGT environment */
```

# Clocking Scheme

The ABFM communicates with the DUT through a parallel interface. The ABFM requires a clock input which matches the parallel data rate of the DUT. The same clock that drives the DUT should be sent as the input clock to the BFM. This clock is directly used as the internal clock of the ABFM as shown in Figure B-4.

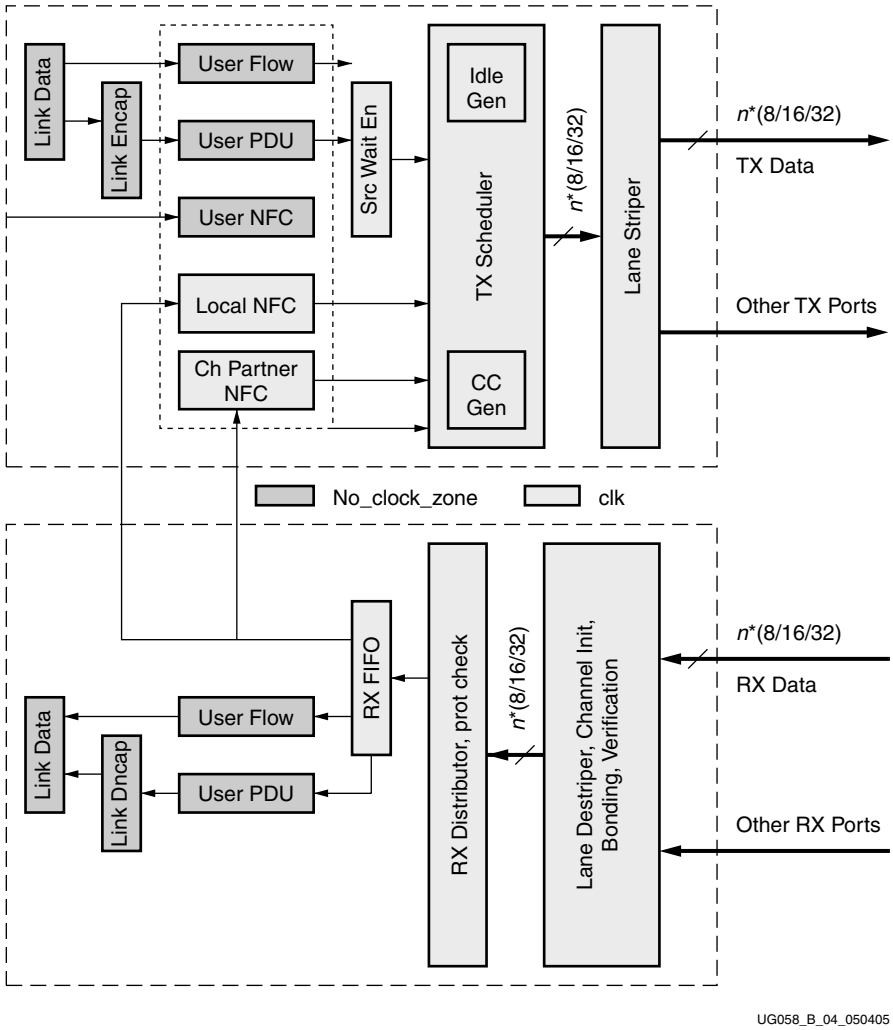


*Figure B-4:* **ABFM Parallel Interface: Clock Zones**

## PLI Interface and Test Bench Integration

A new set of function calls are described for the parallel interface. The functionality of these system calls are identical to the serial interface system calls (see "System Calls" in Chapter 2 for details). All the system/function calls are listed below in order with explanation of differences wherever applicable.

### $bfm_initialize(id,no_of_inputs,no_of_outputs,in_port1,..,in_portn,out_port1,..,out_portm,arg1,..argx);

The rules to order the ports are described in detail in "System Calls" in Chapter 2. The following example for a 804 design illustrates how the wrapper for the BFM needs to be written. The BFM package includes a file called `abfmp_tester_pli.v` which illustrates this as follows:

```
    inputclk;
    inputrst;
    input  [0:63] rx_data;
    input  [0:7]  rx_char_is_k;
    input  [0:7]  rx_char_is_comma;
    input  [0:7]  rx_not_in_table;
    input  [0:7]  rx_disp_err;
    input  [0:3]  rx_buf_status;
    input  [0:11] rx_clk_cor_cnt;
    input  [0:3]  rx_polarity_in;
    input  [0:3]  tx_buf_err;
    input  [0:7]  tx_k_err;
    input  [0:3]  chbonddone;
    output [0:63] tx_data;
    output [0:7]  tx_char_is_k_em;
    output [0:3]  tx_reset_em;
    output [0:3]  en_comma_align_em;
    output [0:3]  enmcommaalign_em;
    output [0:3]  enpcommaalign_em;
    output [0:3]  enchansync_em;
    output [0:7]  tx_char_is_comma_em;
    output [0:7]  tx_not_in_table_em;
    output [0:7]  tx_disp_err_em;
    output [0:3]  chbonddone_out_em;
    output [0:3]  rx_polarity_em;
    output [0:3]  rx_realign_em;

reg  tx_fifo_full_ud,tx_fifo_full_uf,tx_fifo_full_nf;
reg  rx_fifo_full_ud,rx_fifo_full_uf,rx_fifo_full_nf;
reg  rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf;

reg  [0:7]conf_monitor;
reg  [0:12]   conf_result;
reg  [0:6]init_monitor;
```

Hence, the system call in the case of parallel interface is specified as follows:

```
$bfm_initialize(id,MODE,13,13, clk, rst,
rx_data,rx_char_is_k,rx_char_is_comma, rx_not_in_table, rx_disp_err,
rx_buf_status,rx_clk_cor_cnt,rx_polarity_in,tx_buf_err,
tx_k_err,chbondone,
tx_data,tx_char_is_k_em, tx_reset_em,en_comma_align_em,enmcommaalign_em,
enpcommaalign_em,enchansync_em,tx_char_is_comma_em,tx_not_in_table_em,
tx_disp_err_em,chbonddone_out_em,rx_polarity_em,
tx_fifo_full_ud, tx_fifo_full_uf, tx_fifo_full_nf, rx_fifo_full_ud,
rx_fifo_full_uf, rx_fifo_full_nf
rx_fifo_empty_ud, rx_fifo_empty_uf,
rx_fifo_empty_nf,rx_pkts_in_fifo_ud,rx_pkts_in_fifo_uf,rx_pkts_in_fifo_nf
conf_monitor, conf_result, init_monitor);
```

These statements enable accurate port mapping between the ABFM and the test bench.

$bfm_execute(id)

See "$bfm_execute(id)" in Chapter 2.

$bfm_read_file_ud(id,"filename")

See "$bfm_read_file_ud(id,"filename")" in Chapter 2.

$bfm_read_file_uf(id,"filename")

See "$bfm_read_file_uf(id,"filename")" in Chapter 2.

$bfm_read_file_nf(id,"filename")

See "$bfm_read_file_nf(id,"filename")" in Chapter 2.

$bfm_write_file_ud(id,"filename")

See "$bfm_write_file_ud(id,"filename")" in Chapter 2.

$bfm_write_file_uf(id,"filename")

See "$bfm_write_file_ud(id,"filename")" in Chapter 2.

$bfm_write_file_nf(id,"filename")

See "$bfm_write_file_nf(id,"filename")" in Chapter 2.

$bfm_send_next_pkt_ud(id,no_of_pkts)

See "$bfm_send_next_pkt_ud(id,no_of_pkts)" in Chapter 2.

$bfm_send_next_pkt_uf(id,no_of_pkts)

See "$bfm_send_next_pkt_uf(id,no_of_pkts)" in Chapter 2.

$bfm_send_next_pkt_nf(id,no_of_pkts)

See "$bfm_send_next_pkt_nf(id,no_of_pkts)" in Chapter 2.

$bfm_get_next_pkt_ud(id,no_of_pkts)

See "$bfm_get_next_pkt_ud(id,no_of_pkts)" in Chapter 2.

$bfm_get_next_pkt_uf(id,no_of_pkts)

See "$bfm_get_next_pkt_uf(id,no_of_pkts)" in Chapter 2.

$bfm_get_next_pkt_nf(id,no_of_pkts)

See "$bfm_get_next_pkt_nf(id,no_of_pkts)" in Chapter 2.

### $bfm_change_param(id,param1,val,..,paramn,val)

The usage of the system call and majority of the parameter definitions are the same as described in "$bfm_change_param(id,param1,val,param2..,paramn,valn)" in Chapter 2, except for the following differences.

| Argument | Value(s) | Default Value | Description |
|---|---|---|---|
| tx_polarity | 0,1 | 0 | 0=Normal operation<br>1=Invert tx bytes |
| rx_polarity | 0,1 | 0 | 0=Normal operation<br>1=Invert rx bytes |

### $bfm_get_simulation_time(id)

See "$bfm_get_simulation_time(id)" in Chapter 2.

### $bfm_reset(id)

See "$bfm_reset(id)" in Chapter 2.

### $bfm_pulse_event (id,"sig name","width_str","freq_str","val_str",…)

See "$bfm_pulse_event(id,"sig name","width_str","freq_str","val_str",...);" in Chapter 2.

## FLI Interface and Test Bench Integration

The functionality of the system calls are exactly identical to serial interface system calls (see "System Calls" in Chapter 2). The only difference between serial interface and parallel interface system calls is in the entity declaration of bfm_initialize which needs to be modified to include additional ports for the parallel interface. The BFM package includes the example file `abfmp_tester_fli.v` which illustrates the changes.

## VHPI Interface and Test Bench Integration

The functionality of the system calls are exactly identical to serial interface system calls (see "System Calls" in Chapter 2). The only difference between serial interface and parallel interface system calls is in the foreign subprogram declaration and usage of bfm_initialize and bfm_execute. Both subprograms need to include additional ports in their prototype declaration. The BFM package contains example files `abfmp_tester_vhpi.vhd` and `abfmp_vhpi_subprograms.vhd` to illustrate the changes.

## Limitations

The parallel BFM does not inherit all features of the serial BFM. The limitations are:

- The parallel BFM does not support automatic compliance. However continuous conformance provides protocol checking in exactly the same way as a serial interface.
- PCS layer is not implemented in the parallel BFM. The user needs to implement the PCS layer in the test environment while using a transceiver without a PCS layer. However, Xilinx MGTs do implement the PCS layer and do not require additional development in the test environment.