Mentor®
A Siemens Business

Mentor VIP – Intel FPGA Edition AMBA AXI3 and AXI4 User Guide

Software Version 2019.1

February 2019

Note - Viewing PDF files within a web browser causes some links not to function (see MG595892).
Use HTML for full navigation

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: mentor.com
Support Center: support.mentor.com

Send Feedback on Documentation: support.mentor.com/doc_feedback_form

Table of Contents

List of Examples

List of Figures

List of Tables

Preface

About This User Guide

This user guide describes the AXI3/AXI4 application interface (API) of the Mentor® Verification IP – Intel® FPGA Edition (Verification IP – Intel FPGA Edition) and how it conforms to the *AMBA® AXI™ and ACE™ Protocol Specification*, AXI3™, AXI4™, and AXI4-Lite™, ACE, and ACE-Lite™ Issue E (ARM IHI 0022E).

Note

This release supports only the AMBA AXI3, AXI4, AXI4-Lite, and AXI4-Stream™ protocols. The AMBA ACE protocol is not supported in this release.

AMBA AXI Protocol Specification

Mentor VIP – Intel FPGA Edition conforms to the *AMBA® AXI™ and ACE™ Protocol Specification*, AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™ Issue E (ARM IHI 0022E). For restrictions to this protocol, refer to the section Protocol Restrictions.

This user guide refers to this specification as the “AXI Protocol Specification.”

Protocol Restrictions

Mentor VIP – Intel FPGA Edition supports all but the following features of this AXI Protocol Specification, which gives you a simplified API to create desired protocol stimulus.

BFM Dependencies Between Handshake Signals

Starting a write data phase before its write address phase in a transaction is not supported. However, starting a write data phase simultaneously with its write address phase is supported.

The above statement disallowing a write data phase to start before its write address phase in a transaction modifies the AXI3 Protocol Specification write transaction handshake dependencies diagram, Figure A3-6 in Section A3.3.1, by effectively adding double-headed arrows between AWVALID to WVALID and AWREADY to WVALID, with the provision that they can be simultaneous.

The above statement disallowing a write data phase to start before its write address phase in a transaction modifies the AXI4 Protocol Specification slave write response handshake

dependencies diagram, Figure A3-7 in Section A3.3.1, by effectively adding double-headed arrows between AWVALID to WVALID and AWREADY to WVALID, with the provision that they can be simultaneous.

AXI3 BFM Write Data Interleaving

The ability of a BFM to interleave write data is not supported. Therefore, a write data burst that has started will complete before another write data burst with the same or different transaction ID can start. An AXI3 BFM modifies the AXI Protocol Specification by removing Section A5.3.3 concerning the interleaving of write data with different AWID signal values.

BFM Read Data Interleaving

The ability of a BFM to interleave read data is not supported. Therefore, a read data burst that has started will complete before another read data burst with the same or different transaction ID can start. A BFM modifies the AXI Protocol Specification by changing the following statement in Section A5.3.1 concerning the interleaving of read data with different ARID signal values.

Read data of transactions with different ARID values *cannot* be interleaved.

License Requirements

Note

A license is required to access the Mentor VIP – Intel FPGA Edition Bus Functional Models (BFMs) and Inline Monitor.

- To access the Mentor VIP – Intel FPGA Edition and upgrade to the Quartus Prime Subscription Edition software version 18.1 from a previous version, you must regenerate your license file.
- To access Mentor VIP – Intel FPGA Edition with the Quartus Prime Web Edition software, you must upgrade to version 18.1 and purchase a Mentor VIP – Intel FPGA Edition seat license by contacting your Intel sales representative.

Supported Simulators

Mentor VIP – Intel FPGA Edition supports the following simulators:

- Mentor Graphics Questa SIM and ModelSim 10.6d
- Synopsys® VCS® and VCS-MX N-2017.12-SP2 on Linux
- Cadence® Xcelium™ 18.03 on Linux

Simulator GCC Requirements

Mentor VIP requires that the installation directory of the simulator includes the GCC libraries shown in Table 1. If the installation of the GCC libraries was an optional part of the simulator's installation and the Mentor VIP does not find these libraries, an error message displays similar to the following:

```
ModelSim / Questa Sim
# ** Error: (vsim-8388) Could not find the MVC shared library : GCC not
found in installation directory
(/home/user/altera2/<version>/modelsim_ase) for platform "linux".  Please
install GCC version "gcc-5.3.0-linux"
```

Table-1. Simulator GCC Requirements

Simulator	Version	GCC Version(s)	Search Path
Mentor Questa SIM			
	10.6d	5.3.0 (Linux 32 bit)	<install dir>/gcc-5.3.0-linux
		5.3.0 (Linux 64 bit)	<install dir>/gcc-5.3.0-linux_x86_64
		4.2.1 (Windows 32 bit)	<install dir>/gcc-4.2.1-mingw32vc12
Mentor ModelSim			
	10.6d	5.3.0 (Linux 32 bit)	<install dir>/gcc-5.3.0-linux
		5.3.0 (Linux 64 bit)	<install dir>/gcc-5.3.0-linux_x86_64
		4.2.1 (Windows 32 bit)	<install dir>/gcc-4.2.1-mingw32vc12
Synopsys® VCS®/VCS-MX			
	N-2017.12-SP2	4.8.3 (Linux 32/64 bit)	$VCS_HOME/gnu/linux/gcc-4.8.3
			$VCS_HOME/gnu/gcc-4.8.3

Note: If you set the environment variable VG_GNU_PACKAGE, then it is used instead of the VCS_HOME environment variable.

Cadence® Xcelium™			
	18.03	4.8 (Linux 32/64 bit)	<install dir>/tools/cdsgcc/gcc/4.8

Note: Use the *cds_tools.sh* executable to find the Incisive installation. Ensure $PATH includes the installation path and *<install dir>/tools/cdsgcc/gcc/4.8/install/bin*. Also, ensure the LD_LIBRARY_PATH includes *<install dir>/tools/cdsgcc/gcc/4.8/install/lib*.

AXI3 and AXI4 Syntax References

Throughout this user guide, the syntax *axi* or *axi4* is used when an AXI3 or AXI4 protocol argument is referenced: *axi* is used for AXI3 protocol arguments; *axi4* is used for AXI4 protocol arguments. Uppercase AXI3 and AXI4 are used for enumerated arguments.

When a task is applicable to both AXI3 and AXI4 protocols, either a single (*) or double-asterisk (**) is used in the syntax description. A single asterisk is used for nonenumerated arguments; a double-asterisk is used for enumerated arguments.

Chapter 1
Mentor VIP – Intel FPGA Edition

Mentor VIP – Intel FPGA Edition provides BFMs to simulate the behavior and to facilitate IP verification. It includes the following interfaces:

- AXI3 BFM with master, slave, and inline monitor interfaces
- AXI4 BFM with master, slave, and inline monitor interfaces

Advantages of Using BFMs and Monitors

Using the Mentor VIP – Intel FPGA Edition has the following advantages:

- Accelerates the verification process by providing key verification test bench components
- Provides BFM components that implement the AMBA AXI Protocol Specification, which serves as a reference for the protocol
- Provides a full suite of configurable assertion checking in each BFM

Implementation of BFMs

The Mentor VIP – Intel FPGA Edition BFMs, master, slave, and inline monitor components are implemented in SystemVerilog. Also included are wrapper components so that you can use the BFMs in VHDL verification environments with simulators that support mixed-language simulation.

Mentor VIP – Intel FPGA Edition provides a set of APIs for each BFM that you can use to construct, instantiate, control, and query signals in all BFM components. Your test programs must use only these public access methods and events to communicate with each BFM. To ensure support in current and future releases, your test programs must use the standard set of APIs to interface with the BFMs. Nonstandard APIs and user-generated interfaces cannot be supported in future releases.

The test program drives the stimulus to the DUTs and determines whether the behavior of the DUTs is correct by analyzing the responses. The BFMs translate the test program stimuli (transactions), creating the signaling for the AMBA AXI Protocol Specification. The BFMs also check for protocol compliance by firing an assertion when a protocol error is observed.

What Is a Transaction?

A transaction for Mentor VIP – Intel FPGA Edition represents an instance of information that is transferred between a master and a slave peripheral, and that adheres to the protocol used to transfer the information. For example, a write transaction transfers an address phase, a data burst, followed by a response phase. A subsequent instance of transferred information requires a new and unique transaction.

Each transaction has a dynamic Transaction Record that exists for the life of the transaction. The life of a transaction record starts when it is created and ends when the transaction completes. The transaction record is automatically discarded when the transaction ends.

When created, a transaction contains *transaction fields* that you set to define two transaction aspects:

- *Protocol fields* are transferred over the protocol signals.
- *Operation fields* determine how the information is transferred and when the transfer is complete.

For example, a write transaction record holds the *protection* information in the *prot* protocol field; the value of this field is transferred over the AWPROT protocol signals during an address phase. A write transaction also has a *transaction_done* operation field that indicates when the transaction is complete; this field is not transferred over the protocol signals. These two types of transaction fields, protocol and operation, establish a dynamic record during the life of the transaction.

In addition to transaction fields, you specify *arguments* to tasks, functions, and procedures that permit you to create, set, and get the dynamic transaction record during the lifetime of a transaction. Each BFM has an API that controls how you access the BFM transaction record. How you access the record also depends on the source code language, whether it is VHDL or SystemVerilog. Methods for accessing transactions based on the language you use are explained in detail in the relevant chapters of this user guide.

AXI Transactions

Note

The following description of an AXI transaction is applicable to AXI3 and AXI4 protocols.

A complete read/write transaction transfers information between a master and a slave peripheral. Transaction fields described in “What Is a Transaction?” on page 24 determine what is transferred and how information is transferred. During the lifetime of a transaction, the roles of the master and slave ensure that a transaction completes successfully and that transferred information adheres to the protocol specification. Information flows in both directions during a

transaction with the master initiating the transaction, and the slave reporting back to the master that the transaction has completed.

An AXI protocol uses five channels (three write channels and two read channels) to transfer protocol information. Each of these channels has a pair of handshake signals, *VALID and *READY, that indicates valid information on a channel and the acceptance of the information from the channel.

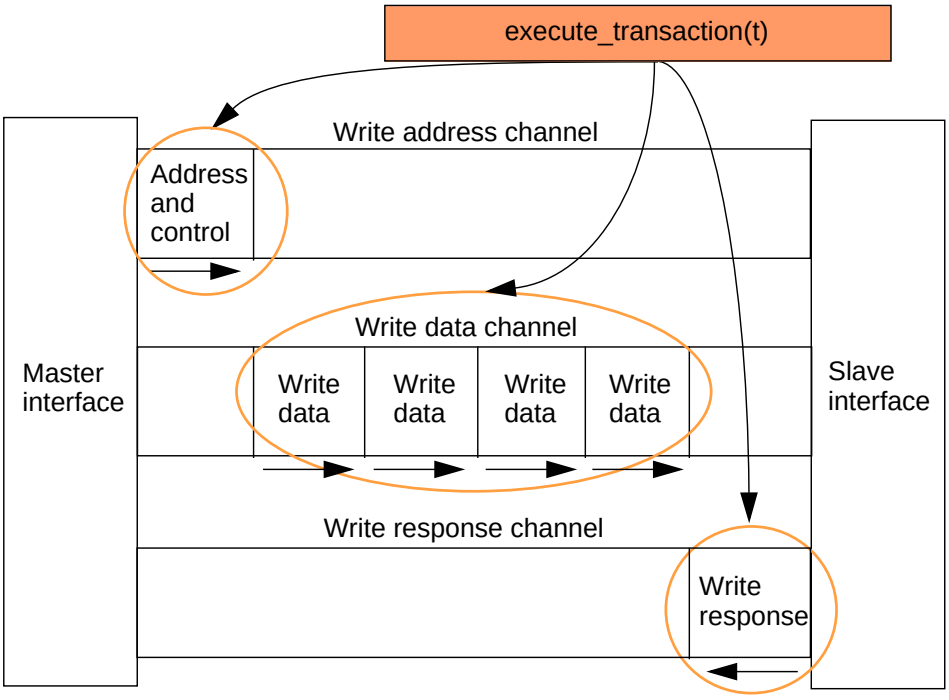
AXI Write Transaction Master and Slave Roles

Note

The following description of a write transaction references SystemVerilog BFM API tasks. There are equivalent VHDL BFM API procedures that perform the same functionality.

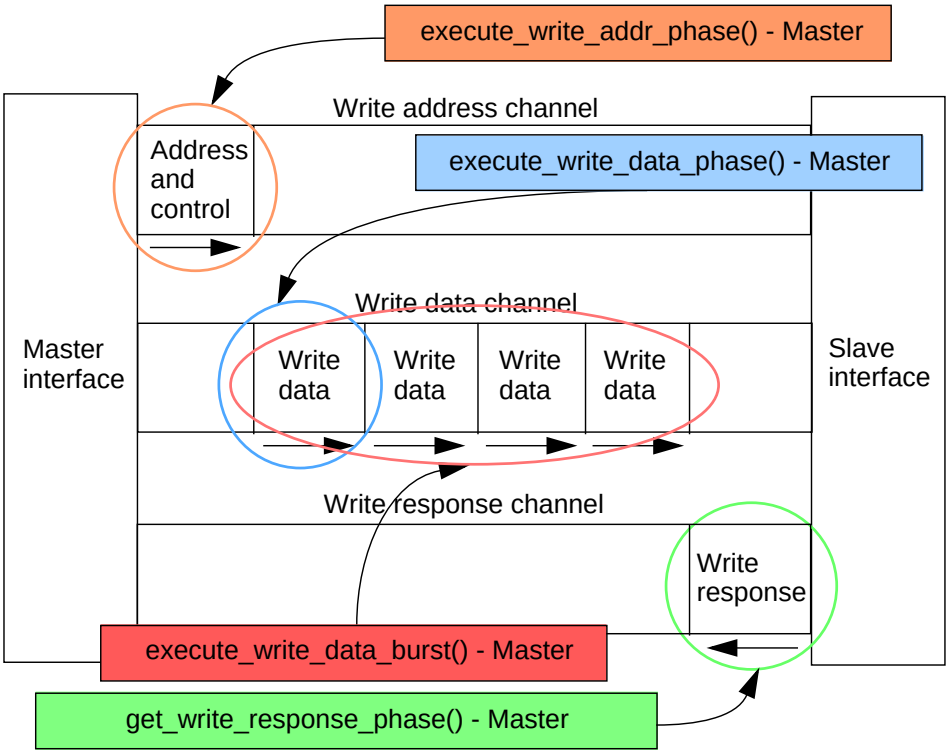
For a write transaction, the master calls the *create_write_transaction()* task to define the information to be transferred and then calls the *execute_transaction()* task to initiate the transfer of information as the following figure illustrates.

Figure 1-1. Execute Write Transaction



The *execute_transaction()* task results in the master calling the *execute_write_addr_phase()* task followed by the *execute_write_data_burst()* task . The *execute_write_data_burst()* calls the *execute_write_data_phase()* task for each phase (beat) of the burst defined by a *burst_length* transaction field as illustrated in Figure 1-2.

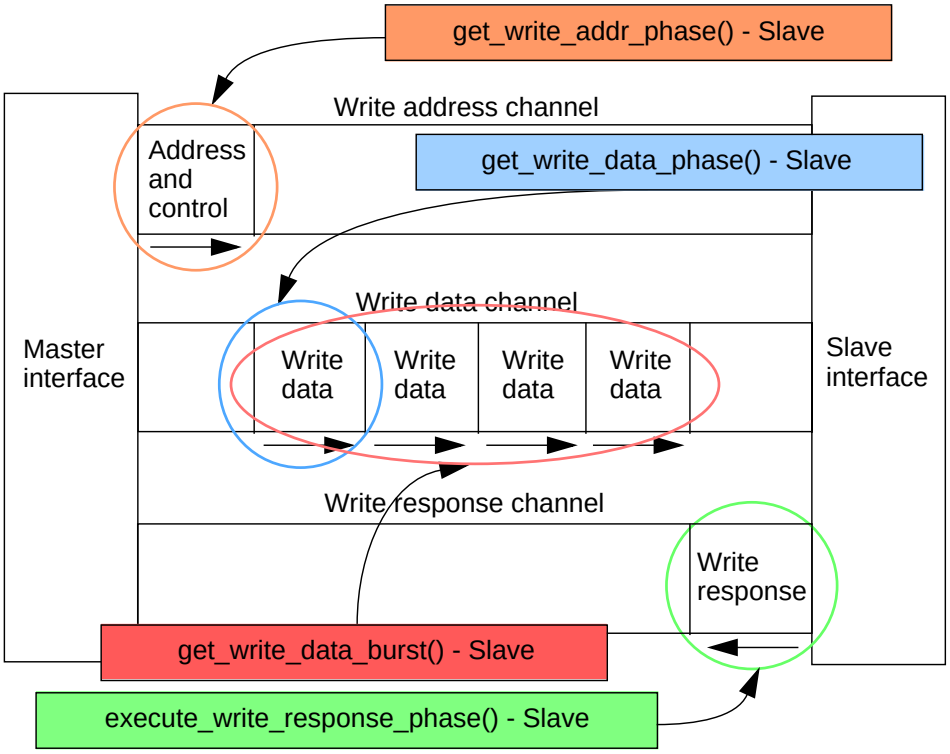
Figure 1-2. Master Write Transaction Phases



The master then calls the *get_write_response_phase()* task to receive the response from the slave and to complete its role in the write transaction.

The slave also creates a transaction by calling the *create_slave_transaction()* task to accept the transfer of information from the master. The address phase and data burst are received by the slave calling the *get_write_addr_phase()* task, followed by the *get_write_data_burst()* task. The *get_write_data_burst()* calls the *get_write_data_phase()* task for each phase (beat) of the burst defined by a *burst_length* transaction field, as illustrated in Figure 1-3.

Figure 1-3. Slave Write Transaction Phases



The slave then executes a write response phase by calling the *execute_write_response_phase()* task and completes its role in the write transaction.

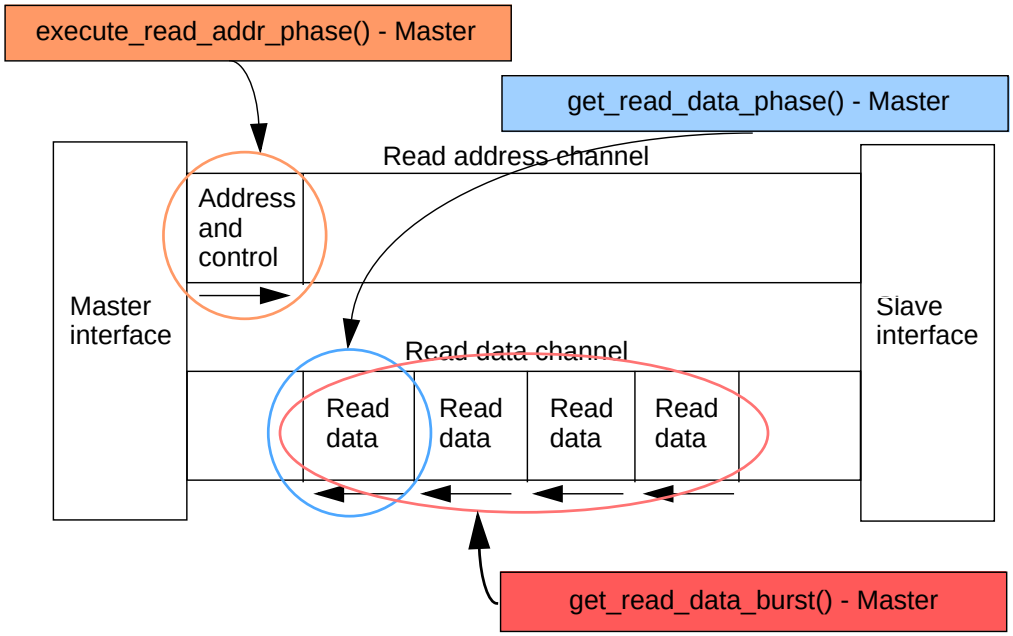
AXI Read Transaction Master and Slave Roles

Note

The following description of a read transaction references the SystemVerilog BFM API tasks. There are equivalent VHDL BFM API procedures that perform the same functionality.

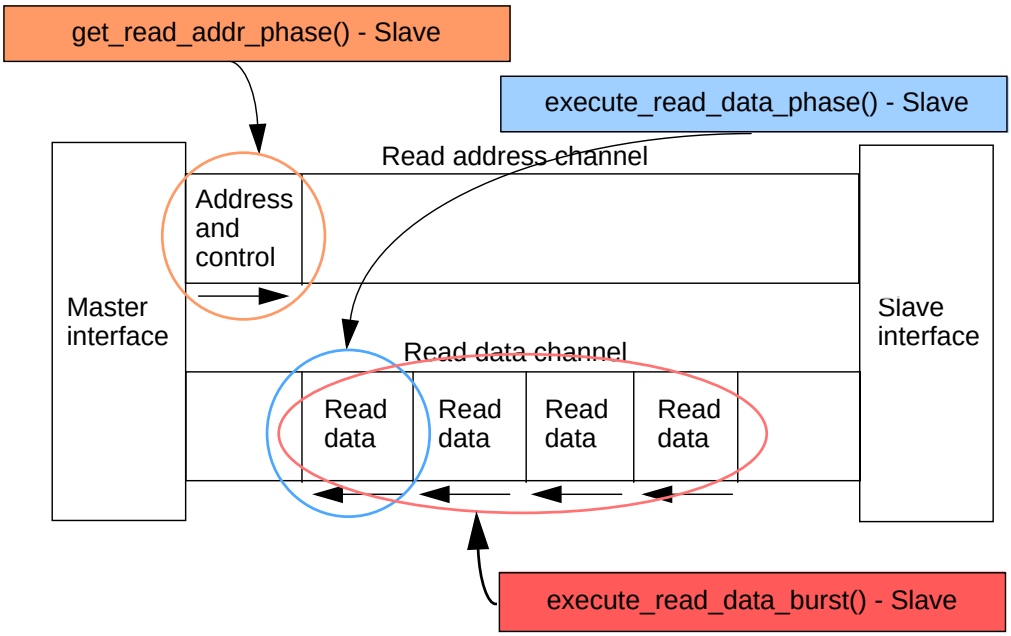
A read transaction is similar to a write transaction. The master initiates the read by calling the *create_read_transaction()* and *execute_transaction()* tasks. The *execute_transaction()* calls the the *execute_read_addr_phase()* task followed by the *get_read_data_burst()* task. The *get_read_data_burst()* calls the *get_read_data_phase()* task for each phase (beat) of the burst defined by a *burst_length* transaction field, as illustrated in Figure 1-4.

Figure 1-4. Master Read Transaction Phases



The slave creates a read transaction by calling the *create_slave_transaction()* task to accept the transfer of read information from the master. The slave accepts the address phase by calling the *get_read_addr_phase()* task, and then executes the data burst by calling the *execute_read_data_burst()* task. The *execute_read_data_burst()* calls the *execute_read_data_phase()* task for each phase (beat) of the burst defined by the *burst_length* transaction field, as illustrated in Figure 1-5.

Figure 1-5. Slave Read Transaction Phases

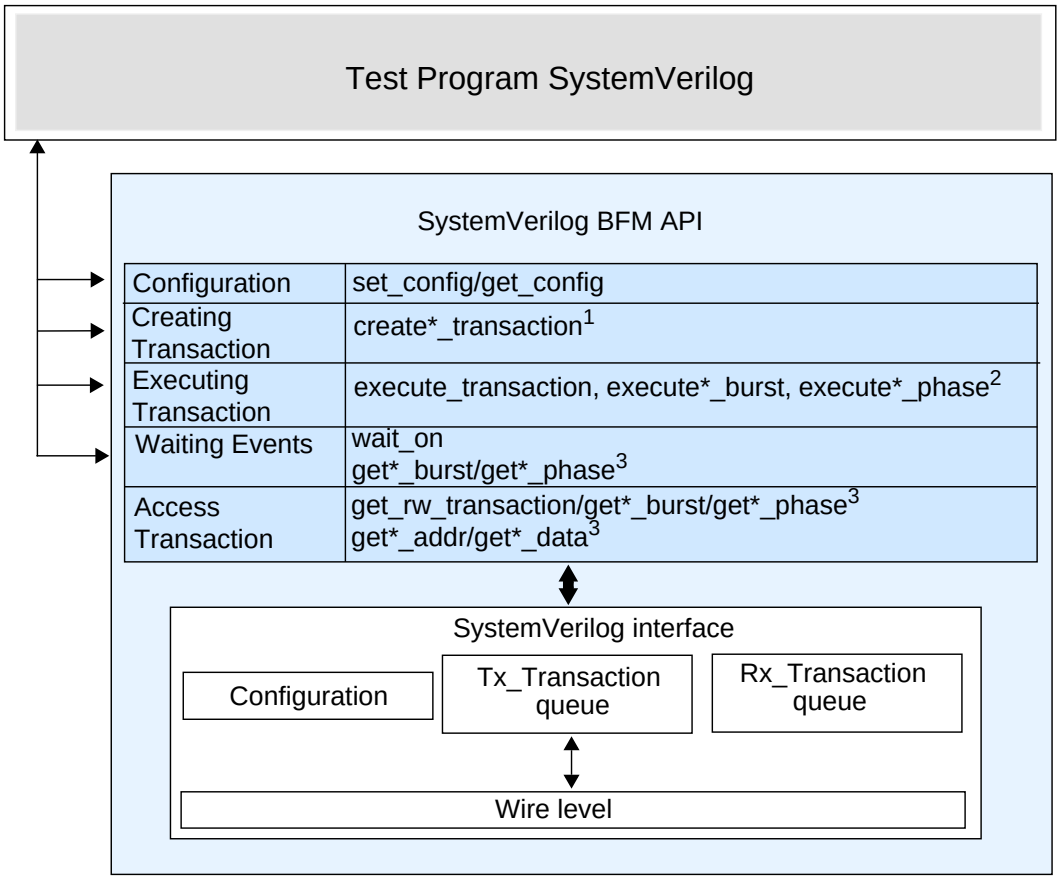


Chapter 2
SystemVerilog API Overview

This chapter provides the functional description of the SystemVerilog (SV) Application Programming Interface (API) for all BFM (master, slave, and monitor) components. For each BFM, you can configure the protocol transaction fields that are executed on the protocol signals, as well as control the operational transaction fields that permit delays to be introduced between the handshake signals for each of the five address, data, and response channels.

In addition, each BFM API has tasks that wait for certain events to occur on the system clock and reset signals, and tasks to get and set information about a particular transaction.

Figure 2-1. SystemVerilog BFM Internal Structure



Notes: 1. Refer to create*_transaction()
2. Refer to execute_transaction(), execute*_burst(), execute*_phase()
3. Refer to get*()

Configuration

Configuration sets timeout delays, error reporting, and other attributes of the BFM. Each BFM has a *set_config()* function that sets the configuration of the BFM. Refer to the individual BFM APIs for details.

Each BFM also has a *get_config()* function that returns the configuration of the BFM. Refer to the individual BFM APIs for details.

set_config()

The following test program code sets the burst timeout factor for a transaction in the master BFM.

```
// Setting the burst timeoutfactor to 1000
master_bfm.set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

Note

This test program code segment is for AXI3 BFMs. Substitute the AXI_CONFIG_BURST_TIMEOUT_FACTOR enumeration with AXI4_CONFIG_BURST_TIMEOUT_FACTOR for AXI4 BFMs.

get_config()

The following test program code gets the protocol signal hold time in the master BFM.

```
 // Getting hold time value
hold_time = master_bfm.get_config(AXI_CONFIG_HOLD_TIME);
```

Note

This test program code segment is for AXI3 BFMs. Substitute the AXI_CONFIG_HOLD_TIME enumeration with AXI4_CONFIG_HOLD_TIME for AXI4 BFMs.

Creating Transactions

To transfer information between a master BFM and a slave DUT over the protocol signals, you must create a transaction in the master test program. Similarly, to transfer information between a master DUT and a slave BFM, you must create a transaction in the slave test program. To monitor the transfer of information using a monitor BFM, you must create a transaction in the monitor test program.

When you create a transaction, a Transaction Record is created and exists for the life of the transaction. This transaction record can be accessed by the BFM test programs during the life of the transaction as it transfers information between the master and slave.

Transaction Record

The transaction record contains two types of transaction fields, *protocol* and *operational*, that either transfer information over the protocol signals or define how and when a transfer occurs.

Protocol fields contain transaction information that is transferred over protocol signals. For example, the *prot* field is transferred over the AWPROT protocol signals during a write transaction.

Operational fields define how and when the transaction is transferred. Their content is not transferred over protocol signals. For example, the *operation_mode* field controls the blocking/nonblocking operation of a transaction, but this information is not transferred over the protocol signals.

AXI3 Transaction Definition

The transaction record exists as a SystemVerilog class definition in each BFM. Example 2-1 shows the definition of the *axi_transaction* class members that form the transaction record.

Example 2-1. AXI3 Transaction Definition

```
// Global Transaction Class
class axi_transaction;
    // Protocol
    bit [((`MAX_AXI_ADDRESS_WIDTH) - 1):0]  addr;
    axi_size_e size;
    axi_burst_e burst;
    axi_lock_e lock;
    axi_cache_e cache;
    axi_prot_e prot;
    bit [((`MAX_AXI_ID_WIDTH) - 1):0]  id;
    bit [3:0] burst_length;
    bit [(((`MAX_AXI_RDATA_WIDTH > `MAX_AXI_WDATA_WIDTH) ?
`MAX_AXI_RDATA_WIDTH : `MAX_AXI_WDATA_WIDTH)) - 1):0] data_words [];
    bit [(((`MAX_AXI_WDATA_WIDTH / 8)) - 1):0] write_strobes [];
    axi_response_e resp[];
    bit [7:0] addr_user;
    axi_rw_e read_or_write;
    int address_valid_delay;
    int data_valid_delay[];
    int write_response_valid_delay;
    int address_ready_delay;
    int data_ready_delay[];
    int write_response_ready_delay;
// Housekeeping
    bit gen_write_strobes = 1'b1;
    axi_operation_mode_e  operation_mode  = AXI_TRANSACTION_BLOCKING;
    axi_delay_mode_e      delay_mode      = AXI_VALID2READY;
    axi_write_data_mode_e write_data_mode = AXI_DATA_WITH_ADDRESS;
    bit data_beat_done[];
    bit transaction_done;
...
endclass
```

AXI4 Transaction Definition

The transaction record exists as a SystemVerilog class definition in each BFM. Example 2-2 shows the definition of the *axi4_transaction* class members that form the transaction record.

Example 2-2. AXI4 Transaction Definition

```
// Global Transaction Class
class axi4_transaction;
    // Protocol
    axi4_rw_e read_or_write;
    bit [((`MAX_AXI4_ADDRESS_WIDTH) - 1):0]  addr;
    axi4_prot_e prot;
    bit [3:0] region;
    axi4_size_e size;
    axi4_burst_e burst;
    axi4_lock_e lock;
    axi4_cache_e cache;
    bit [3:0] qos;
    bit [((`MAX_AXI4_ID_WIDTH) - 1):0] id;
    bit [7:0] burst_length;
    bit [((`MAX_AXI4_USER_WIDTH) - 1):0]  addr_user;
    bit [(((`MAX_AXI4_RDATA_WIDTH > `MAX_AXI4_WDATA_WIDTH) ?
`MAX_AXI4_RDATA_WIDTH : `MAX_AXI4_WDATA_WIDTH)) - 1):0] data_words [];
    bit [(((`MAX_AXI4_WDATA_WIDTH / 8)) - 1):0] write_strobes [];
    axi4_response_e resp[];
    int address_valid_delay;
    int data_valid_delay[];
    int write_response_valid_delay;
    int address_ready_delay;
    int data_ready_delay[];
    int write_response_ready_delay;

    // Housekeeping
    bit gen_write_strobes = 1'b1;
    axi4_operation_mode_e  operation_mode  = AXI4_TRANSACTION_BLOCKING;
    axi4_write_data_mode_e write_data_mode = AXI4_DATA_WITH_ADDRESS;
    bit data_beat_done[];
    bit transaction_done;

...

endclass
```

Note

This *axi4_transaction* class code is shown for information only. Access to each transaction record during its life is performed by various *set*()* and *get*()* tasks described later in this chapter.

The contents of the transaction record are defined in Table 2-1.

Table 2-1. Transaction Fields

Transaction Field	Description
Protocol Transaction Fields	
addr	A bit vector (the length is equal to the ARADDR/AWADDR signal bus width) containing the starting *address* of the first transfer (beat) of a transaction. The *addr* value is transferred over the ARADDR or AWADDR signals for a read or write transaction, respectively.
prot	An enumeration containing the *protection* type of a transaction. The types of *protection* are: **_NORM_SEC_DATA (default) **_PRIV_SEC_DATA **_NORM_NONSEC_DATA **_PRIV_NONSEC_DATA **_NORM_SEC_INST **_PRIV_SEC_INST **_NORM_NONSEC_INST **_PRIV_NONSEC_INST The *prot* value is transferred over the ARPROT or AWPROT signals for a read or write transaction, respectively.
region	(AXI4) A 4-bit vector containing the *region* identifier of a transaction. The *region* value is transferred over the ARREGION or AWREGION signals for a read or write transaction, respectively.
size	An enumeration to hold the *size* of a transaction. The types of *size* are: **_BYTES_1 **_BYTES_2 **_BYTES_4 **_BYTES_8 **_BYTES_16 **_BYTES_32 **_BYTES_64 **_BYTES_128 The *size* value is transferred over the ARSIZE or AWSIZE signals for a read or write transaction, respectively.

Table 2-1. Transaction Fields (cont.)

Transaction Field	Description
burst	An enumeration to hold the *burst* of a transaction. The types of *burst* are: **_FIXED **_INCR **_WRAP **_BURST_RSVD The *burst* value is transferred over the ARBURST or AWBURST signals for a read or write transaction, respectively.
lock	An enumeration to hold the *lock* of a transaction. The types of *lock* are: **_NORMAL **_EXCLUSIVE (AXI3) AXI_LOCKED (AXI3) AXI_LOCKED_RSVD The *lock* value is transferred over the ARLOCK or AWLOCK signals for a read or write transaction, respectively.
cache	(AXI3) An enumeration to hold the *cache* of a transaction. The types of *cache* are: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW; The *cache* value is transferred over the ARCACHE or AWCACHE signals for a read or write transaction, respectively.

Table 2-1. Transaction Fields (cont.)

Transaction Field	Description
cache	(AXI4) An enumeration to hold the *cache* of a transaction. The types of *cache* are: AXI4_NONMODIFIABLE_NONBUF AXI4_BUF_ONLY AXI4_CACHE_NOALLOC AXI4_CACHE_2 AXI4_CACHE_3 AXI4_CACHE_RSVD4 AXI4_CACHE_RSVD5 AXI4_CACHE_6 AXI4_CACHE_7 AXI4_CACHE_RSVD8 AXI4_CACHE_RSVD9 AXI4_CACHE_10 AXI4_CACHE_11 AXI4_CACHE_RSVD12 AXI4_CACHE_RSVD13 AXI4_CACHE_14 AXI4_CACHE_15 The *cache* value is transferred over the ARCACHE or AWCACHE signals for a read or write transaction, respectively.
qos	(AXI4) A 4-bit vector to hold the *Quality of Service* (qos) identifier of a transaction. The *qos* value is transferred over the ARQOS or AWQOS signals for a read or write transaction, respectively.
id	A bit vector (of length equal to the *A*RID/AWID signal bus width) that holds the identification tag of a transaction. The id value is transferred over the AWID/BID signals for a write transaction and over the ARID/RID signals for a read transaction.
burst_length	A 4-bit (8-bit for AXI4) vector to hold the burst length of a transaction. The *burst_length* value is transferred over the ARLEN or AWLEN signals for a read or write transaction, respectively.
addr_user	A bit vector (of length equal to the ARUSER/AWUSER signal bus width) to hold the address channel *user data* of a transaction. The *addr_data* value is transferred over the ARUSER or AWUSER signals for a read or write transaction, respectively.

Table 2-1. Transaction Fields (cont.)

Transaction Field	Description
data_words	An unsized array of bit vectors (of length equal to the greater of the RDATA/WDATA signal bus widths) to hold the *data words* of the payload. A *data_words* array element is transferred over the RDATA or WDATA signals per beat of the read or write data channel, respectively.
write_strobes	An unsized array of bit vectors (of length equal to the WDATA signal bus width divided by 8) to hold the write strobes. A *write_strobes* array element is transferred over the WSTRB signals per beat of the write data channel.
resp	An unsized enumeration array to hold the responses of a transaction. The types of *response* are: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR; A *resp* array element is transferred over the RRESP signals per beat of the read data channel, and over the BRESP signals for a write transaction, respectively.
Operational Transaction Fields	
read_or_write	An enumeration to hold the *read or write* control flag. The types of *read_or_write* are: **_TRANS_READ **_TRANS_WRITE
address_valid_delay	An integer to hold the delay value of the address channel AWVALID and ARVALID signals (measured in ACLK cycles) for a read or write transaction, respectively.
data_valid_delay	An unsized array of integers to hold the delay values of the data channel WVALID and RVALID signals (measured in ACLK cycles) for a read or write transaction, respectively.
write_response_valid_delay	An integer to hold the delay value of the write response channel BVALID signal (measured in ACLK cycles) for a write transaction.
address_ready_delay	An integer to hold the delay value of the address channel AWREADY and ARREADY signals (measured in ACLK cycles) for a read or write transaction, respectively.

Table 2-1. Transaction Fields (cont.)

Transaction Field	Description
data_ready_delay	An unsized array of integers to hold the delay values of the data channel WREADY and RREADY signals (measured in ACLK cycles) for a read or write transaction, respectively.
write_response_ready_delay	An integer to hold the delay value of the write response channel BREADY signal (measured in ACLK cycles) for a write transaction.
gen_write_strobes	Automatically correct write strobes flag. Refer to Automatic Generation of Byte Lane Strobes for details.
operation_mode	An enumeration to hold the *operation mode* of the transaction. The two types of *operation_mode* are: **_TRANSACTION_NON_BLOCKING **_TRANSACTION_BLOCKING
delay_mode	(AXI3) An enumeration to hold the *delay mode* control flag. The types of *delay_mode* are: AXI_VALID2READY AXI_TRANS2READY Refer to AXI3 BFM Delay Mode for details.
write_data_mode	An enumeration to hold the *write data mode* control flag. The types of *write_data_mode* are: **_DATA_AFTER_ADDRESS The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.

Table 2-1. Transaction Fields (cont.)

Transaction Field	Description
data_beat_done	An unsized bit array to hold the *done* flag for each beat in a read or write data burst when it has completed.
transaction_done	A bit to hold the *done* flag for a transaction when it has completed.

The master BFM API allows you to create a master transaction by providing only the address and burst length arguments for a read or write transaction. All other protocol transaction fields automatically default to legal protocol values to create a complete master transaction record. Refer to the *create_read_transaction()* and *create_write_transaction()* functions for default protocol read and write transaction field values.

The slave BFM API allows you to create a slave transaction without providing any arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the *create_slave_transaction()* function for default protocol transaction field values.

The monitor BFM API allows you to create a monitor transaction without providing any arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the *create_monitor_transaction()* function for default protocol transaction field values.

Note

If you change the default value of a protocol transaction field, this value is valid for all future transactions until you set a new value.

create*_transaction()

There are two master BFM API functions available to create transactions, *create_read_transaction()* and *create_write_transaction()*, a *create_slave_transaction()* for the slave BFM API, and a *create_monitor_transaction()* for the monitor BFM API.

For example, the following master BFM test program creates a simple write transaction with a start address of 1, and a single data phase with a data value of 2, the master BFM test program would contain the following code:

```
// Define a variable trans of type axi_transaction
axi_transaction write_trans;

// Create master write transaction
write_trans = bfm.create_write_transaction(1);
write_trans.data_words[0]   = 2;
```

For example, to create a simple slave transaction, the slave BFM test program contains the following code:

```
// Define a variable slave_trans of type axi_transaction
axi_transaction slave_trans;

// Create slave transaction
slave_trans = bfm.create_slave_transaction();
```

Note

These test program code segments are for AXI3 BFMs. Substitute the *axi_transaction* type definition with *axi4_transaction* for AXI4 BFMs.

Executing Transactions

Executing a transaction in a master/slave BFM test program initiates the transaction onto the protocol signals. Each master/slave BFM API has execution tasks that push transactions into the BFM internal transaction queues. Figure 2-1 on page 31 illustrates the internal BFM structure.

execute_transaction(), execute*_burst(), execute*_phase()

If the DUT is a slave, then the *execute_transaction()* task is called in the master BFM test program. If the DUT is a master, then the execute*_burst() and execute*_phase() tasks are called in the slave BFM test program.

For example, to execute a master write transaction the master BFM test program contains the following code:

```
// By default the execution of a transaction will block
bfm.execute_transaction(write_trans);
```

For example, to execute a slave write response phase, the slave BFM test program contains the following code:

```
// By default the execution of a transaction will block
bfm.execute_write_response_phase(slave_trans);
```

Waiting Events

Each BFM API has tasks that block the test program code execution until an event has occurred.

The *wait_on()* task blocks the test program until an ACLK or ARESETn signal event has occurred before proceeding.

The *get*_transaction(), get*_burst(), get*_phase(), get*_cycle()* tasks block the test program code execution until a complete transaction, burst, phase, or cycle has occurred, respectively.

wait_on()

A BFM test program can wait for the positive edge of the ARESETn signal using the following code:

```
// Block test program execution until the positive edge of the clock
bfm.wait_on(AXI_RESET_POSEDGE);
```

Note

These test program code segments are for AXI3 BFMs. Substitute the AXI_RESET_POSEDGE enumeration with AXI4_RESET_POSEDGE for AXI4 BFMs.

get*_transaction(), get*_burst(), get*_phase(), get*_cycle()

A slave BFM test program can use a received write address phase to form the response to the write transaction. The test program gets the write address phase for the transaction by calling the *get_write_addr_phase()* task. This task blocks until it has received the address phase, allowing the test program to call the *execute_write_response_phase()* task for the transaction at a later stage, as shown in the slave BFM test program in Example 2-3.

Example 2-3. Slave Test Program Using get_write_addr_phase()

```
slave_trans = bfm.create_slave_transaction();
bfm.get_write_addr_phase(slave_trans);

...

bfm.execute_write_response_phase(slave_trans);
```

Note

Not all BFM APIs support the full complement of *get*_transaction(), get*_burst(), get*_phase(), get*_cycle()* tasks. Refer to the individual master, slave, or monitor BFM API for details.

Access Transaction Record

Each BFM API has tasks that can access a complete or partially complete Transaction Record. The *set*()* and *get*()* tasks are used in a test program to set and get information from the transaction record.

Note

The *set*()* and *get*()* tasks are not explicitly described in each BFM API chapter. The simple rule for the task name is *set_* or *get_* followed by the name of the transaction field accessed. Refer to "Transaction Fields" on page 35 for transaction field name details.

set*()

For example, to set the WSTRB write strobes signal for the first phase (beat) in the Transaction Record of a write transaction, the master test program would use the *set_write_strobes()* task, as shown in the following code:

```
write_trans.set_write_strobes(4'b0010, 0);
```

get*()

For example, a slave BFM test program uses a received write address phase to get the AWPROT signal value from the Transaction Record, as shown in the following slave BFM test program code:

```
// Define a variable prot_value of type axi_transaction
axi_prot_e prot_value;

slave_trans = bfm.create_slave_transaction();

// Wait for a write address phase
bfm.get_write_addr_phase(slave_trans);

... …

// Get the AWPROT signal value of the slave transaction
prot_value = bfm.get_prot(slave_trans);
```

Note

These test program code segments are for AXI3 BFMs. For AXI4 BFMs; substitute the *axi_transaction* type definition with *axi4_transaction* and *axi_prot_e* with *axi4_prot_e*.

Operational Transaction Fields

Operational transaction fields control the way a transaction is executed onto the protocol signals. They also indicate when a data phase (beat) or transaction is complete.

Automatic Generation of Byte Lane Strobes

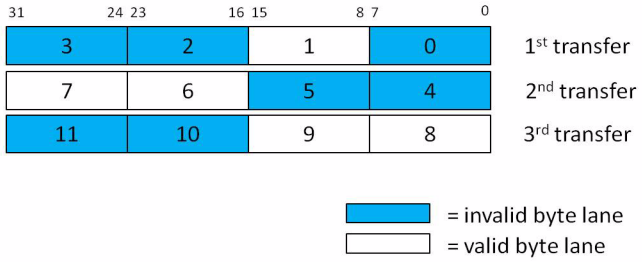
The master BFM permits unaligned and narrow write transfers by using byte lane strobe (WSTRB*)* signals to indicate which byte lanes contain valid data per data phase (beat).

When you create a write transaction in your master BFM test program, the *write_strobes* variable is available to store the write strobe values for each write data phase (beat) in the transaction. To assist you in creating the correct byte lane strobes, automatic correction of any previously set *write_strobes* is performed by default during execution of the write transaction, or write data phase (beat). You can disable this default behavior by setting the operational transaction field *gen_write_strobes = 0,* which allows any previously set *write_strobes* to pass through uncorrected onto the protocol WSTRB signals. In this mode, with the automatic correction disabled, you are responsible for setting the correct *write_strobes* for the whole transaction.

The automatic correction algorithm performs a bit-wise AND operation on any previously set *write_strobes*. To do the corrections, the correction algorithm uses the equations described in the AMBA AXI Protocol Specification, Section A3.4.1, that define valid write data byte lanes for legal protocol. Therefore, if you require automatic generation of all *write_strobes*, before the write transaction executes, you must set all *write_strobes* to 1, indicating that all bytes lanes initially contain valid write data prior to the execution of the write transaction. Automatic correction then sets the relevant *write_strobes* to 0 to produce legal protocol WSTRB signals.

For example, Figure 2-2 shows byte lanes that can contain valid data for a write transaction with a starting address = 0x01, size = 0b001 (2 bytes), type = INCR, and the length = 0b0010 (3 beats) for a 32-bit write data bus.

Figure 2-2. Valid Data on Byte Lanes During a Write Transaction



In the above example, if you set all *write_strobes[]* array elements to 1 before executing the write transaction, automatic correction produces the following results while the transaction executes.

	Prior to Execution		During Execution
1st data phase	write_strobes[0]=0b1111	->	write_strobes[0]=0b0010
2nd data phase	write_strobes[1]=0b1111	->	write_strobes[1]=0b1100
3rd data phase	write_strobes[2]=0b1111	->	write_strobes[2]=0b0011

If you randomly set all *write_strobes[]* array elements to 0 or 1, before executing the write transaction, automatic correction corrects *only* those *write_strobes[]* array elements that were previously set to 1, as shown below.

	Prior to Execution		During Execution
1st data phase	write_strobes[0]=0b1010	->	write_strobes[0]=0b0010
2nd data phase	write_strobes[1]=0b1010	->	write_strobes[1]=0b1000
3rd data phase	write_strobes[2]=0b1010	->	write_strobes[2]=0b0010

Note

To automatically generate all WSTRB signals for a write transaction, set all *write_strobes[]* array elements to 1 before executing the write transaction or write data burst.

Operation Mode

By default, each read or write transaction performs a blocking operation which prevents a following transaction from starting until the current active transaction completes.

You can configure this behavior to be nonblocking by setting the *operation_mode* transaction field to the enumerate type value AXI_TRANSACTION_NON_BLOCKING instead of the default AXI_TRANSACTION_BLOCKING.

For example, in a master BFM test program you create a transaction by calling the *create_read_transaction()* or *create_write_transaction()* tasks, which creates a transaction record. Before executing the transaction record, you can change the *operation_mode* as follows:

```
// Create a write transaction to create a transaction record
trans = bfm.create_write_transaction(1);

// Change operation_mode to be nonblocking in the transaction record
trans.operation_mode(AXI_TRANSACTION_NON_BLOCKING);
```

Note

These test program code segments are for AXI3 BFMs. Substitute the AXI_TRANSACTION_NON_BLOCKING enumeration with AXI4_TRANSACTION_NON_BLOCKING for AXI4 BFMs.

Channel Handshake Delay

Each of the five protocol channels have *VALID and *READY handshake signals that control the rate at which information is transferred between a master and slave. The API to control these handshake signals differs between the AXI3 BFMs and AXI4 BFMs. Refer to the AXI3 BFM Handshake Delay and AXI3 BFM Delay Mode for details of the AXI3 BFM API, and AXI4 BFM Handshake Delay for details of the AXI4 BFM API.

AXI3 BFM Handshake Delay

You can configure the delay between the *VALID and *READY handshake signals for each of the five protocol channels. You can define the delay per phase (beat) basis for a particular transaction, measured from the positive edge of ACLK when *VALID is asserted. You can also set the delay set from the completion of a previous transaction phase (*VALID and *READY both asserted).

AXI3 BFM Handshake Signal Delay Transaction Fields

The transaction record contains transaction fields to configure the desired handshake delay pattern for a particular transaction phase on any of the five protocol channels. The master BFM configures the *VALID and *READY signal delays that it asserts, and the slave BFM configures the *VALID and *READY signal delays that it asserts. Table 2-2 specifies which operational delay transaction fields are configured by the master and slave BFMs.

Table 2-2. Handshake Signal Delay Transaction Fields

Signal	Operational Transaction Field	Configuration BFM
AWVALID	address_valid_delay	Master
AWREADY	address_ready_delay	Slave
WVALID	data_valid_delay	Master
WREADY	data_ready_delay	Slave
BVALID	write_response_valid_delay	Slave
BREADY	write_response_ready_delay	Master
ARVALID	address_valid_delay	Master
ARREADY	address_ready_delay	Slave

Table 2-2. Handshake Signal Delay Transaction Fields (cont.)

Signal	Operational Transaction Field	Configuration BFM
RVALID	data_valid_delay	Slave
RREADY	data_ready_delay	Master

Note

The data channel handshake signal transaction fields (*data_valid_delay[]* and *data_ready_delay[]*) are defined as arrays so that the *VALID to *READY delay can be configured on a per data phase (beat) basis in a transaction.

AXI4 BFM Handshake Delay

The delay between the *VALID and *READY handshake signals for each of the five protocol channels is controlled in a BFM test program using *execute_*_ready()*, *get_*_ready()*, and *get_*_cycle()* tasks. The *execute_*_ready()* tasks place a value onto the *READY signals and the *get_*_ready()* tasks retrieve a value from the *READY signals. The *get_*_cycle()* tasks wait for a *VALID signal to be asserted and are used to insert a delay between the *VALID and *READY signals in the BFM test program.

For example, the master BFM test program code below inserts a specified delay between the read channel RVALID and RREADY handshake signals using the *execute_read_data_ready()* and *get_read_data_cycle()* tasks.

```
// Set the RREADY signal to '0' so that it is nonblocking
fork
   bfm.execute_read_data_ready(1'b0);
join_none

// Wait until the RVALID signal is asserted and then wait_on the specified
// number of ACLK cycles
bfm.get_read_data_cycle;
repeat(5) bfm.wait_on(AXI4_CLOCK_POSEDGE);

// Set the RREADY signal to '1' so that it blocks for an ACLK cycle
bfm.execute_read_data_ready(1'b1);
```

AXI4 BFM *VALID Signal Delay Transaction Fields

The transaction record contains a *_*valid_delay* transaction field for each of the five protocol channels to configure the delay value prior to the assertion of the *VALID signal for the channel. The master BFM holds the delay configuration for the *VALID signals that it asserts, and the slave BFM holds the delay configuration for the

*VALID signals that it asserts.
Table 2-3 specifies which *_*valid_delay* fields are configured by the master and slave BFMs.

Table 2-3. Master and Slave *_valid_delay Configuration Fields

Signal	Operational Transaction Field	Configuration BFM
AWVALID	address_valid_delay	Master
WVALID	data_valid_delay	Master
BVALID	write_response_valid_delay	Slave
ARVALID	address_valid_delay	Master
RVALID	data_valid_delay	Slave

Note

In the transaction record, the data channel handshake signal transaction field (*data_valid_delay[]*) is defined as an array, which allows you to configure the *VALID delay on a per data phase (beat) basis in a transaction.

AXI4 BFM *READY Handshake Signal Delay Transaction Fields

The transaction record contains a *_*ready_delay* transaction field for each of the five protocol channels to store the delay value that occurred between the assertion of the *VALID and *READY handshake signals for the channel. Table 2-4 specifies the *_*ready_delay* field corresponding to the *READY signal delay.

Table 2-4. Master and Slave *_ready_delay Transaction Fields

Signal	Operational Transaction Field
AWREADY	address_ready_delay
WREADY	data_ready_delay
BREADY	write_response_ready_delay
ARREADY	address_ready_delay
RREADY	data_ready_delay

Note

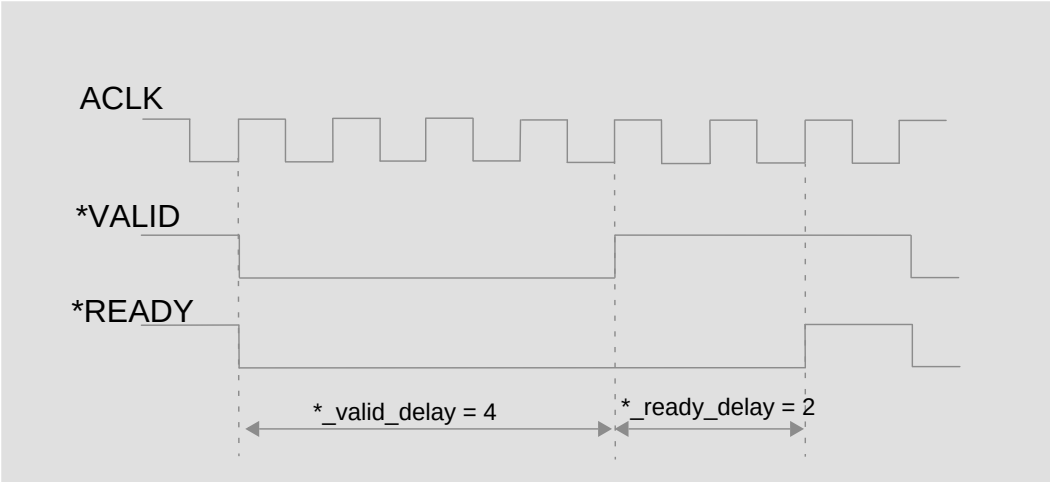
In the transaction record, the data channel handshake signal transaction field (*data_ready_delay[]*) is defined as an array so that the *READY delay can be recorded on a per data phase (beat) basis in a transaction.

AXI3 BFM Delay Mode

You can configure the delay mode c on a per transaction basis using the *delay_mode* operational transaction field. You can configure this transaction field to the enumerated type values of AXI_VALID2READY (default) or AXI_TRANS2READY.

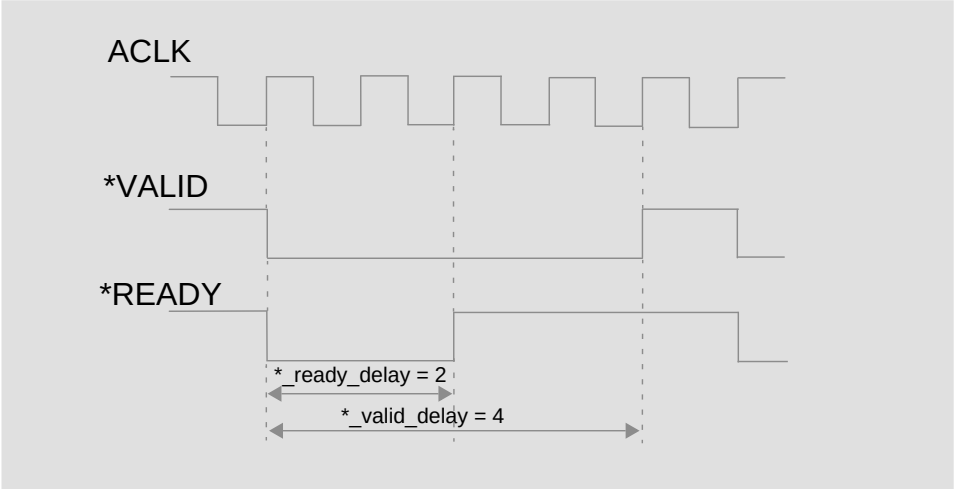
The default configuration (*delay_mode* = AXI_VALID2READY) corresponds to the delay measured from the positive edge of ACLK when *VALID is asserted in a transaction. Figure 2-3 demonstrates how to achieve a *VALID asserted before *READY handshake.

Figure 2-3. Operational Transaction Field delay_mode = AXI_VALID2READY



The other configuration (*delay_mode* = AXI_TRANS2READY) corresponds to the delay measured from the completion of a previous transaction phase (*VALID and *READY both asserted). Figure 2-4 demonstrates how to achieve a *READY before *VALID handshake.

Figure 2-4. Operational Transaction Field delay_mode = AXI_TRANS2READY



Data Beat Done

There is a *data_beat_done* transaction field for each transaction, defined as an array, to indicate when each data phase (beat) has completed. Each element of the *data_beat_done* array is set to 1 when each data phase (beat) has completed in a data burst.

You call the *get_read_data_phase()* task in the master BFM test program to investigate how many beats of a read data burst have completed by analyzing how many elements of the *data_beat_done* array have been set to 1. Similarly, the *get_write_data_phase()* task can be called in the slave BFM test program to analyze a write data burst.

Transaction Done

The *transaction_done* field in each transaction indicates when the transaction is complete.

In a master BFM test program, you call the *get_read_data_burst()* task to investigate whether a read transaction is complete, and the *get_write_response_phase()* to investigate whether a write transaction is complete.

Chapter 3
SystemVerilog AXI3 and AXI4 Master BFMs

This chapter provides information about the SystemVerilog AXI3 and AXI4 master BFMs. Each BFM has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the lifetime of the transaction.

Note

Due to AXI3 protocol specification changes, for some BFM tasks, you reference the AXI3 BFM by specifying AXI instead of AXI3.

Master BFM Protocol Support

The AXI3 master BFM supports the AMBA AXI3 protocol with restrictions described in "Protocol Restrictions" on page 19. In addition to the standard protocol, it supports user sideband signals AWUSER and ARUSER.

The AXI4 master BFM supports the AMBA AXI4 protocol with restrictions described in "Protocol Restrictions" on page 19.

Master Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following master BFM API timing and events.

The AMBA AXI Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP as a result of these directives, declarations, options, or initialization files:

- `timescale directives in design elements
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options

- Local or site-wide simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. The recommended practice is to use timeunit and timeprecision declarations. For details, refer to Section 3.14, “System Time Units and Precision,” of the *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800™-2012 , February 21, 2013. This user guide refers to this document as the *IEEE Standard for SystemVerilog*.

Master BFM Configuration

A master BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters that configure the widths of the address, ID and data signals, and transaction fields to specify timeout factors, slave exclusive support, setup and hold times, and so on.

You can change the address, ID and data signal widths from their default settings by assigning them new values, usually in the top-level module of the test bench. These new values are then passed to the master BFM using a parameter port list of the master BFM module. For example, the code extract below shows the AXI3 master BFM with the address, ID and data signal widths defined in *module top()* and passed to the *master_test_program* parameter port list:

```
module top ();

   parameter AXI_ADDRESS_WIDTH = 24;
   parameter AXI_RDATA_WIDTH = 16;
   parameter AXI_WDATA_WIDTH = 16;
   parameter AXI_ID_WIDTH = 4;

   master_test_program #(AXI_ADDRESS_WIDTH, AXI_RDATA_WIDTH,
AXI_WDATA_WIDTH, AXI_ID_WIDTH) bfm_master(....);
```

Table 3-1 lists parameter names for the address, ID, data signals, etc, and their default values.

Table 3-1. Master BFM Signal Width Parameters

Signal Width Parameter (Note: ** = AXI or AXI4)	**Description**
**_ADDRESS_WIDTH	Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
**_RDATA_WIDTH	Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_WDATA_WIDTH	Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.

Table 3-1. Master BFM Signal Width Parameters (cont.)

Signal Width Parameter (Note: ** = AXI or AXI4)	Description
**_ID_WIDTH	ID signal width in bits. This applies to the RID and WID signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
AXI4_USER_WIDTH	(AXI4) User data signal width in bits. This applies to the ARUSER, AWUSER, RUSER, WUSER and BUSER signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 8.
AXI4_REGION_MAP_SIZE	(AXI4) Region signal width in bits. This applies to the ARREGION and AWREGION signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 16.
index	Ignored for the SystemVerilog master BFM.
READ_ISSUING_ CAPABILITY	The maximum number of outstanding read transactions that can be issued from the master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
WRITE_ISSUING_ CAPABILITY	The maximum number of outstanding write transactions that can be issued from the master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
COMBINED_ISSUING_ CAPABILITY	The maximum number of outstanding combined read and write transactions that can be issued from the master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
USE_*	(AXI4) Each protocol signal connection to the master BFM can be enabled or disabled. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. 0 = disabled. 1 = enabled (default).

A master BFM has configuration fields that you can set with the *set_config()* function to configure timeout factors, slave exclusive support, and setup and hold times, and so on. You can

also get the value of a configuration field using the *get_config()* function. Table 3-2 describes the full list of configuration fields.

Table 3-2. Master BFM Configuration

Configuration Field (Note: ** = AXI or AXI4)	**Description**
Timing Variables	
**_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.1 Default: 0.
**_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.1 Default: 0.
**_CONFIG_MAX_TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
**_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
**_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY	The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods. Default: 1000.
**_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY	The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY	The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY	The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY	The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods. Default 10000.

Table 3-2. Master BFM Configuration (cont.)

Configuration Field (Note: ** = AXI or AXI4)	**Description**
Master Attributes	
AXI4_CONFIG_ENABLE_RLAST	(AXI4) Configures the support for the optional RLAST signal. 0 = disabled 1 = enabled (default)
Slave Attributes	
**_CONFIG_SUPPORT_EXCLUSIVE_ ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_UNDER_ RESET	(AXI3) The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
**_CONFIG_SLAVE_START_ADDR	Configures the start address map for the slave.
**_CONFIG_SLAVE_END_ADDR	Configures the end address map for the slave.
**_CONFIG_READ_DATA_REORDERING_ DEPTH	The slave read reordering depth. Refer to the AMBA AXI Protocol Specification for more details. Default: 1.
Error Detection	
**_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
**_CONFIG_ENABLE_ASSERTION	Individual enable/disable of assertion check in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to Master Timing and Events for details of simulator time-steps.

Master Assertions

Each master BFM performs protocol error checking using the built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

AXI3 Assertion Configuration

By default, all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI_CONFIG_ENABLE_ASSERTION);

// Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

Note
Do not confuse the AXI_CONFIG_ENABLE_ASSERTION bit vector with the AXI_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI3 assertions, refer to “AXI3 Assertions” on page 725.

AXI4 Assertion Configuration

By default, all built-in assertions are enabled in the master AXI4 BFM. To globally disable them in the master BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI4_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI4_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

Note

For a complete listing of AXI4 assertions, refer to “AXI4 Assertions” on page 738.

SystemVerilog Master API

This section describes the SystemVerilog master API.

set_config()

This function sets the configuration of the master BFM.

Prototype

```
// * = axi | axi4
function void set_config
(
   input *_config_e config_name,
   input *_max_bits_t config_val
);
```

Arguments config_name

(AXI3) Configuration name:

AXI_CONFIG_SETUP_TIME
AXI_CONFIG_HOLD_TIME
AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
AXI_CONFIG_BURST_TIMEOUT_FACTOR
AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
AXI_CONFIG_MASTER_WRITE_DELAY
AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
AXI_CONFIG_ENABLE_ALL_ASSERTIONS
AXI_CONFIG_ENABLE_ASSERTION
AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
AXI_CONFIG_READ_DATA_REORDERING_DEPTH
AXI_CONFIG_SLAVE_START_ADDR
AXI_CONFIG_SLAVE_END_ADDR
AXI_CONFIG_MASTER_ERROR_POSITION
AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

(AXI4) Configuration name:

AXI4_CONFIG_SETUP_TIME
AXI4_CONFIG_HOLD_TIME
AXI4_CONFIG_BURST_TIMEOUT_FACTOR
AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
AXI4_CONFIG_ENABLE_RLAST
AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4_CONFIG_ENABLE_ASSERTION
AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
AXI4_CONFIG_ENABLE_QOS
AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
AXI4_CONFIG_SLAVE_START_ADDR
AXI4_CONFIG_SLAVE_END_ADDR

See “Master BFM Configuration” on page 52 for descriptions and valid values.

Returns None

AXI3 Example

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1);
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

AXI4 Example

```
set_config(AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE, 1);
set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the master BFM.

Prototype

```
// * = axi | axi4
function void get_config
(
   input *_config_e config_name,
);
```

Arguments

config_name

(AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

(AXI4) Configuration name:

- AXI4_CONFIG_SETUP_TIME
- AXI4_CONFIG_HOLD_TIME
- AXI4_CONFIG_BURST_TIMEOUT_FACTOR
- AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI4_CONFIG_ENABLE_RLAST
- AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
- AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI4_CONFIG_ENABLE_ASSERTION
- AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI4_CONFIG_ENABLE_QOS
- AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI4_CONFIG_SLAVE_START_ADDR
- AXI4_CONFIG_SLAVE_END_ADDR

Returns

config_val

See “Master BFM Configuration” on page 52 for descriptions and valid values.

AXI3 Example

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS);
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR);
```

AXI4 Example

```
get_config(AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE);
get_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_write_transaction()

This nonblocking function creates a write transaction with a start address *addr* and optional *burst_length* arguments. All other transaction fields default to legal protocol values, unless previously assigned a value. It returns with the **_transaction* record.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function automatic *_transaction create_write_transaction
(
    input bit [((**_ADDRESS_WIDTH) - 1):0]  addr,
    bit [3:0] burst_length = 0 // optional
);
```

Arguments

addr	Start address
burst_length	(Optional) Burst length. Default: 0.

Protocol Transaction Fields

size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;
lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
cache	(AXI3) Burst cache: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;

	cache	(AXI4) Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;
	prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
	id	Burst ID
	data_words	Data words array.
	write_strobes	Write strobes array: Each strobe 0 or 1.
	resp	Burst response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;
	region	(AXI4) Region identifier.
	qos	(AXI4) Quality-of-Service identifier.
	addr_user	Address channel user data.
	data_user	(AXI4) Data channel user data.
	resp_user	(AXI4) Response channel user data.
Operational Transaction Fields	gen_write_strobes	Generate write strobes flag: 0 = user supplied write strobes. 1 = auto-generated write strobes (default).
	operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING; (default)
	delay_mode	(AXI3) Delay mode: AXI_VALID2READY; (default) AXI_TRANS2READY;

	write_data_mode	Write data mode: **_DATA_AFTER_ADDRESS; The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS; (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.
Operational Transaction Fields	address_valid_delay	Address channel AWVALID delay measured in ACLK cycles for this transaction (default = 0).
	data_valid_delay	Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
	write_response_ ready_delay	Write response channel BREADY delay measured in ACLK cycles for this transaction (default = 0).
	data_beat_done	Write data channel beat *done* flag array for this transaction.
	transaction_done	Write transaction *done* flag for this transaction.
Returns	The *_*transaction* record.	

AXI3 Example

```
// Create a write transaction with a data burst length of 3 (4 beats) to
// start address 16.
trans = bfm.create_write_transaction(16, 3);
trans.set_size = (AXI_BYTES_4);
trans.set_data_words = ('hACE0ACE1, 0);
trans.set_data_words = ('hACE2ACE3, 1);
trans.set_data_words = ('hACE4ACE5, 2);
trans.set_data_words = ('hACE6ACE7, 3);
```

AXI4 Example

```
// Create a write transaction with a data burst length of 3 to start
// address 16.
trans = bfm.create_write_transaction(16, 3);
trans.set_size = (AXI4_BYTES_4);
trans.set_data_words = ('hACE0ACE1, 0);
trans.set_data_words[= ('hACE2ACE3, 1);
trans.set_data_words = ('hACE4ACE5, 2);
trans.set_data_words = ('hACE6ACE7, 3);
```

create_read_transaction()

This nonblocking function creates a read transaction with a start address *addr* and optional *burst_length* arguments. All other transaction fields default to legal AXI protocol values, unless previously assigned a value. It returns the **_transaction* record.

Prototype

```
// * = axi | axi4
// ** = AXI| AXI4
function automatic *_transaction create_read_transaction
(
    input bit [((**_ADDRESS_WIDTH) - 1):0]  addr,
    bit [3:0] burst_length = 0 //optional
);
```

Arguments

addr	Start address
burst_length	(Optional) Burst length. Default: 0.

Protocol Transaction Fields

size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;
lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
cache	(AXI4) Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;

	prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
		Burst ID
	data_words	Data words array.
	resp	Burst response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;
Operational Transaction Fields	operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING; (default)
	delay_mode	(AXI3) Delay mode: AXI_VALID2READY; (default) AXI_TRANS2READY;
	address_valid_delay	Address channel ARVALID delay measured in ACLK cycles for this transaction (default = 0).
	data_ready_delay	Read data channel RREADY delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
	data_beat_done	Write data channel beat *done* flag array for this transaction.
	transaction_done	Read transaction *done* flag for this transaction.
Returns	*_transaction	The transaction record:

AXI3 Example

```
// Create a read data burst length of 3 (4 beats) to start address 16.
trans = bfm.create_read_transaction(16, 3);
trans.set_size = (AXI_BYTES_4);
```

AXI4 Example

```
// Read data burst length of 3 to start address 16.
trans = bfm.create_read_transaction(16, 3);
trans.set_size = (AXI4_BYTES_4);
```

execute_transaction()

This task executes a master transaction previously created by the *create_write_transaction()*, or *create_read_transaction()*, functions. The transaction can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

The results of *execute_transaction()* for write transactions varies based on how write transaction fields are set. If the *gen_write_strobes* transaction field is set, *execute_transaction()* automatically corrects any previously set *write_strobes*. However, if the *gen_write_strobes* field is not set, then any previously assigned *write_strobes* will be passed through onto the WSTRB protocol signals, which can result in a protocol violation if not correctly set. Refer to "Automatic Correction of Byte Lane Strobes" on page 214 for more details.

If a write transaction *write_data_mode* field is set to *_DATA_WITH_ADDRESS, *execute_transaction()* calls the *execute_write_addr_phase()* and *execute_write_data_burst()* tasks simultaneously; otherwise, *execute_write_data_burst()* will be called after *execute_write_addr_phase()* so that the write data burst occurs after the write address phase (default). It will then call the *get_write_response_phase()* task to complete the write transaction.

For a read transaction, *execute_transaction()* calls the *execute_read_addr_phase()* task followed by the *get_read_data_burst()* task to complete the read transaction.

Prototype

```
// * = axi | axi4
task automatic execute_transaction
(
   *_transaction trans
);
```

Arguments

trans	The *_*transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the read_trans transaction.
bfm.execute_transaction(read_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the read_trans transaction.
bfm.execute_transaction(read_trans);
```

execute_write_addr_phase()

This task executes a master write address phase previously created by the *create_write_transaction()* function. This phase can be blocking (default) or nonblocking, as defined by the transaction *operation_mode* field.

It sets the AWVALID protocol signal at the appropriate time defined by the transaction *address_valid_delay* field.

Prototype

```
// * = axi | axi4
task automatic execute_write_addr_phase
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write_trans transaction.
bfm.execute_transaction(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write_trans transaction.
bfm.execute_transaction(write_trans);
```

execute_read_addr_phase()

This task executes a master read address phase previously created by the *create_read_transaction()* function. This phase can be blocking (default) or nonblocking, as defined by the transaction *operation_mode* field.

It sets the ARVALID protocol signal at the appropriate time, defined by the transaction *address_valid_delay* field.

Prototype

```
// * = axi | axi4
task automatic execute_read_addr_phase
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the read_trans transaction.
bfm.execute_transaction(read_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the read_trans transaction.
bfm.execute_transaction(read_trans);
```

execute_write_data_burst()

This task executes a write data burst previously created by the *create_write_transaction()* task. This burst can be blocking (default) or nonblocking, defined by the transaction *operation_mode* field.

If the transaction *gen_write_strobes* field is set, this task automatically corrects any previously set *write_strobes* field array elements. If the *gen_write_strobes* field is not set, then any previously assigned *write_strobes* field array elements will be passed through onto the WSTRB protocol signals, which can result in a protocol violation if the WSTRB signals are not correctly set. Refer to "Automatic Correction of Byte Lane Strobes" on page 214 for more details.

It calls the *execute_write_data_phase()* task for each beat of the data burst, with the length of the burst defined by the transaction *burst_length* field.

Prototype

```
// * = axi | axi4
task automatic execute_write_data_burst
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write_trans transaction.
bfm.execute_write_data_burst(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write_trans transaction.
bfm.execute_write_data_burst(write_trans);
```

execute_write_data_phase()

This task executes a write data phase (beat) previously created by the *create_write_transaction()* task. This phase can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

The *execute_write_data_phase()* sets the WVALID protocol signal at the appropriate time defined by the transaction record *data_valid_delay* field and sets the *data_beat_done* array *index* element field to 1 when the phase completes.

AXI3 Example

Prototype

```
// * = axi | axi4
task automatic execute_write_data_phase
(
     *_transaction trans,

     int index = 0, // Optional
     output bit last
     );
```

Arguments

trans	The **_transaction* record.
index	Data phase (beat) number.
last	Flag to indicate that this phase is the last beat of data.

Returns None

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write data phase for the first beat of the
// write_trans transaction.
bfm.execute_write_data_phase(write_trans, 0, last);

// Execute the write data phase for the second beat of the
// write_trans transaction.
bfm.execute_write_data_phase(write_trans, 1, last);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Execute the write data phase for the first beat of the
// write_trans transaction.
bfm.execute_write_data_phase(write_trans, 0, last);

// Execute the write data phase for the second beat of the
// write_trans transaction.
bfm.execute_write_data_phase(write_trans, 1, last);
```

get_read_data_burst()

This blocking task gets a read data burst previously created by the *create_read_transaction()* function.

It calls the *get_read_data_phase()* task for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
// * = axi | axi4
task automatic get_read_data_burst
(
   *_transaction trans
);
```

Arguments trans — The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Get the read data burst for the read_trans transaction.
bfm.get_read_data_burst(read_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Execute the read_trans transaction.
bfm.execute_transaction(read_trans);
```

get_read_data_phase()

This blocking task gets a read data phase previously created by the *create_read_transaction()* task.

AXI3 and AXI4

Note

The *get_read_data_phase()* sets the *RREADY* protocol signal at the appropriate time, defined by the transaction record *data_ready_delay* field and sets the *data_beat_done* array *index* element field to 1 when the phase completes. If this is the last phase (beat) of the burst, then it sets the *transaction_done* field to 1 to indicate the whole read transaction is complete. For AXI3, the *get_read_data_phase()* also sets the RREADY protocol signal at the appropriate time, defined by the transaction record *data_ready_delay* field.

Prototype

```
// * = axi | axi4
task automatic get_read_data_phase
(
   *_transaction trans
   ,
   int index = 0 // Optional
);
```

Arguments

trans	The **_transaction* record.
index	(Optional) Data phase (beat) number.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Get the read data phase for the first beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 0);

// Get the read data phase for the second beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 1);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a read transaction with start address of 0 and assign
// it to the local read_trans variable.
read_trans = bfm.create_read_transaction(0);

....

// Get the read data phase for the first beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 0);

// Get the read data phase for the second beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 1);
```

get_write_response_phase()

This blocking task gets a write response phase previously created by the *create_write_transaction()* task.

Note

The *get_write_response_phase()* sets the *transaction_done* field to 1 when the transaction completes to indicate the whole transaction is complete. For AXI3, the *get_write_response_phase()* also sets the BREADY protocol signal at the appropriate time, defined by the transaction record *write_response_ready_delay* field.

Prototype

```
// * = axi | axi4
task automatic get_write_response_phase
 (
     *_transaction trans
 );
```

Arguments trans The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Get the write response phase of the write_trans transaction.
bfm.get_write_response_phase(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a write transaction with start address of 0 and assign
// it to the local write_trans variable.
write_trans = bfm.create_write_transaction(0);

....

// Get the write response phase of the write_trans transaction.
bfm.get_write_response_phase(write_trans);
```

get_read_addr_ready()

This blocking AXI4 task returns the value of the read address channel ARREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_read_addr_ready
(
   output bit ready
);
```

Arguments

ready	The value of the ARREADY signal.

Returns

ready

AXI3 BFM

Note

The *get_read_addr_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the ARREADY signal value
bfm.get_read_addr_ready(ready);
```

get_read_data_cycle()

This blocking AXI4 task waits until the read data channel RVALID signal is asserted.

Prototype `task automatic get_read_data_cycle();`
Arguments None
Returns None

AXI3 BFM

Note
The *get_read_data_cycle()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Waits until the read data channel RVALID signal is asserted.
bfm.get_read_data_cycle();
```

get_write_addr_ready()

This blocking AXI4 task returns the value of the write address channel AWREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_write_addr_ready
(
   output bit ready
);
```

Arguments

ready	The value of the AWREADY signal.

Returns None

AXI3 BFM

Note

The *get_write_addr_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the AWREADY signal
bfm.get_write_addr_ready();
```

get_write_data_ready()

This blocking AXI4 task returns the value of the write data channel WREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_write_data_ready
(
   output bit ready
);
```

Arguments

ready	The value of the WREADY signal.

Returns None

AXI3 BFM

Note

The *get_write_data_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the WREADY signal
bfm.get_write_data_ready();
```

get_write_response_cycle()

This blocking AXI4 task waits until the write response channel BVALID signal is asserted.

Prototype `task automatic get_write_response_cycle();`
Arguments None
Returns None

AXI3 BFM

Note
The *get_write_response_cycle()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Wait until the write response channel BVALID signal is asserted.
bfm.get_write_response_cycle();
```

execute_read_data_ready()

This AXI4 task executes a read data ready by placing the *ready* argument value onto the RREADY signal. It will block for one ACLK period.

Prototype

```
task automatic execute_read_data_ready
(
    bit ready
);
```

Arguments

ready	The value to be placed onto the RREADY signal

Returns None

AXI3 BFM

Note

The *execute_read_data_ready()* task is not available in the AXI3 BFM. Use the *get_read_data_phase()* task along with the transaction record *data_ready_delay* field.

AXI4 Example

```
// Assert and deassert the RREADY signal
forever begin
   bfm.execute_read_data_ready(1'b0);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
   bfm.wait_on(AXI4_CLOCK_POSEDGE);

   bfm.execute_read_data_ready(1'b1);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```

execute_write_resp_ready()

This AXI4 task executes a write response ready by placing the *ready* argument value onto the BREADY signal. It will block for one ACLK period.

Prototype

```
task automatic execute_write_resp_ready
(
    bit ready
);
```

Arguments

ready	The value to be placed onto the BREADY signal

Returns None

AXI3 BFM

Note

The *execute_write_resp_ready()* task is not available in the AXI3 BFM. Use the *get_write_response_phase()* task along with the transaction record *write_response_ready_delay* field.

AXI4 Example

```
// Assert and deassert the BREADY signal
forever begin
   bfm.execute_write_resp_ready(1'b0);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
   bfm.wait_on(AXI4_CLOCK_POSEDGE);

   bfm.execute_write_resp_ready(1'b1);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```

wait_on()

This blocking task waits for an event(s) on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
// * = axi | axi4
// ** = AXI| AXI4
task automatic wait_on
(
   *_wait_e phase,
   input int count = 1 //Optional
);
```

Arguments

phase	Wait for: **_CLOCK_POSEDGE **_CLOCK_NEGEDGE **_CLOCK_ANYEDGE **_CLOCK_0_TO_1 **_CLOCK_1_TO_0 **_RESET_POSEDGE **_RESET_NEGEDGE **_RESET_ANYEDGE **_RESET_0_TO_1 **_RESET_1_TO_0
count	(Optional) Wait for a number of events to occur set by *count*. (default = 1)

Returns None

AXI3 Example

```
bfm.wait_on(AXI_RESET_POSEDGE);
bfm.wait_on(AXI_CLOCK_POSEDGE,10);
```

AXI4 Example

```
bfm.wait_on(AXI4_RESET_POSEDGE);
bfm.wait_on(AXI4_CLOCK_POSEDGE,10);
```

Chapter 4
SystemVerilog AXI3 and AXI4 Slave BFMs

This chapter describes the SystemVerilog AXI3 and AXI4 slave BFMs. Each BFM has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the lifetime of the transaction.

Note

Due to AXI3 protocol specification changes, for some BFM tasks, you reference the AXI3 BFM by specifying AXI instead of AXI3.

Slave BFM Protocol Support

This section defines protocol support for various AXI BFMs.

The AXI3 slave BFM supports the AMBA AXI3 protocol with restrictions described in “Protocol Restrictions” on page 19. In addition to the standard protocol, it supports user sideband signals AWUSER and ARUSER. The AXI4 slave BFM supports the AMBA AXI4 protocol with restrictions described in “Protocol Restrictions” on page 19.

Slave Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following slave BFM API timing and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP based on using the directives, declarations, options, and initialization files below:

- `timescale directives in design elements
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options

- Local or site-wide simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. Using timeunit and timeprecision declarations are recommended. Refer to the *IEEE Standard for SystemVerilog*, Section 3.14 for details.

Slave BFM Configuration

The slave BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters you can use to configure the widths of the address, ID and data signals, and transaction fields to configure timeout factors, slave exclusive support, and setup and hold times, and so on.

You can change the address, ID and data signal widths from their default settings by assigning them with new values, usually performed in the top-level module of the test bench. These new values are then passed into the slave BFM using a parameter port list of the slave BFM module. For example, the code extract below shows the AXI3 slave BFM with the address, ID and data signal widths defined in *module top()* and passed in to the *slave_test_program* parameter port list:

```
module top ();

   parameter AXI_ADDRESS_WIDTH = 24;
   parameter AXI_RDATA_WIDTH = 16;
   parameter AXI_WDATA_WIDTH = 16;
   parameter AXI_ID_WIDTH = 4;

   slave_test_program #(AXI_ADDRESS_WIDTH, AXI_RDATA_WIDTH,
AXI_WDATA_WIDTH, AXI_ID_WIDTH) bfm_slave(....);
```

Table 4-1 lists the parameter names for the address, ID and data signals, and their default values.

Table 4-1. Slave BFM Signal Width Parameters

Signal Width Parameter	Description
**_ADDRESS_WIDTH	Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32
**_RDATA_WIDTH	Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_WDATA_WIDTH	Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.

Table 4-1. Slave BFM Signal Width Parameters (cont.)

Signal Width Parameter	Description
**_ID_WIDTH	ID signal width in bits. This applies to the RID and WID signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
AXI4_USER_WIDTH	(AXI4) User data signal width in bits. This applies to the ARUSER, AWUSER, RUSER, WUSER and BUSER signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 8.
AXI4_REGION_MAP_SIZE	(AXI4) Region signal width in bits. This applies to the ARREGION and AWREGION signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 16.
index	Ignored for the SystemVerilog slave BFM.
READ_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding read transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
WRITE_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
COMBINED_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding combined read and write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
USE_*	(AXI4) Each protocol signal connection to the slave BFM can be enabled or disabled. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. 0 = disabled. 1 = enabled (default).

A slave BFM has configuration fields that you can set with the *set_config()* function to configure timeout factors, slave exclusive support, setup and hold times, and so on. You can also get the value of a configuration field via the *get_config()* function.

Table 4-2 describes the full list of configuration fields.

Table 4-2. Slave BFM Configuration

Configuration Field	Description
Timing Variables	
**_CONFIG_SETUP_TIME	The setup time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_MAX_TRANSACTION_ TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_DATA_ TRANSFER	(AXI3) The maximum number of write data beats that the AXI3 BFM can generate as part of write data burst of write transfer. Default: 1024.
**_CONFIG_BURST_TIMEOUT_ FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
**_CONFIG_MAX_LATENCY_ AWVALID_ASSERTION_TO_AWREADY	The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_ ARVALID_ASSERTION_TO_ARREADY	The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_RVALID_ ASSERTION_TO_RREADY	The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_BVALID_ ASSERTION_TO_BREADY	The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_ WVALID_ASSERTION_TO_WREADY	The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods (default 10000).

Table 4-2. Slave BFM Configuration (cont.)

Configuration Field	Description
AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME	(AXI3) The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_WRITE_DELAY	(AXI3) The master BFM applies the AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME value set. 0 = true (default) 1 = false
AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET	(AXI3) The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true
Master Attributes	
AXI4_CONFIG_ENABLE_RLAST	(AXI4) Configures the support for the optional RLAST signal. 0 = disabled 1 = enabled (default)
Slave Attributes	
AXI4_CONFIG_ENABLE_QOS	(AXI4) The master participates in the Quality-of-Service scheme. If a master does not participate, the AWQOS/ARQOS value used in write/read transactions must be b0000.
**_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)

Table 4-2. Slave BFM Configuration (cont.)

Configuration Field	Description
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET	(AXI3) The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
**_CONFIG_SLAVE_START_ADDR	Configures the start address map for the slave.
**_CONFIG_SLAVE_END_ADDR	Configures the end address map for the slave.
**_CONFIG_READ_DATA_REORDERING_DEPTH	The slave read reordering depth. Refer to the AMBA AXI Protocol Specification for more details. Default: 1.
AXI4_CONFIG_MAX_OUTSTANDING_WR	(AXI4) Configures the maximum number of outstanding write requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal AWREADY=0b0 if this value is exceeded. Default = 0.
AXI4_CONFIG_MAX_OUTSTANDING_RD	(AXI4) Configures the maximum number of outstanding read requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal ARREADY=0b0 if this value is exceeded. Default = 0.
AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE	(AXI4) Holds the number of outstanding write phases from the master that can be processed by the slave. Default = 0.
AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE	(AXI4) Holds the number of outstanding read phases to the master that can be processed by the slave. Default = 0.

Table 4-2. Slave BFM Configuration (cont.)

Configuration Field	Description
Error Detection	
**_CONFIG_ENABLE_ALL_ ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
**_CONFIG_ENABLE_ASSERTION	Individual enable/disable of assertion check in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to Slave Timing and Events for details of simulator time-steps.

Slave Assertions

Each slave BFM performs protocol error checking using the built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

AXI3 Assertion Configuration

By default, all built-in assertions are enabled in the slave BFM. To globally disable them in the slave BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI_CONFIG_ENABLE_ASSERTION);

// Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

Note

Do not confuse the AXI_CONFIG_ENABLE_ASSERTION bit vector with the AXI_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI3 assertions, refer to "AXI3 Assertions" on page 725.

AXI4 Assertion Configuration

By default, all built-in assertions are enabled in the slave AXI4 BFM. To globally disable them in the slave BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS,0);
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI4_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI4_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI4 assertions, refer to "AXI4 Assertions" on page 738.

SystemVerilog Slave API

This section describes the SystemVerilog Slave API.

set_config()

This function sets the configuration of the slave BFM.

Prototype

```
// * = axi | axi4
function void set_config
(
   input *_config_e config_name,
   input *_max_bits_t config_val
);
```

Arguments

config_name (AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS
- AXI_CONFIG_MAX_OUTSTANDING_WR
- AXI_CONFIG_MAX_OUTSTANDING_RD

	(AXI4) Configuration name: AXI4_CONFIG_SETUP_TIME AXI4_CONFIG_HOLD_TIME AXI4_CONFIG_BURST_TIMEOUT_FACTOR AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI4_CONFIG_ENABLE_RLAST AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE AXI4_CONFIG_ENABLE_ALL_ASSERTIONS AXI4_CONFIG_ENABLE_ASSERTION AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY AXI4_CONFIG_ENABLE_QOS AXI4_CONFIG_READ_DATA_REORDERING_DEPTH AXI4_CONFIG_SLAVE_START_ADDR AXI4_CONFIG_SLAVE_END_ADDR AXI4_CONFIG_MAX_OUTSTANDING_WR AXI4_CONFIG_MAX_OUTSTANDING_RD AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE
config_val	See Slave BFM Configuration for descriptions and valid values.

Returns None

AXI3 Example

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1);
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

AXI4 Example

```
set_config(AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE, 1);
set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the slave BFM.

Prototype

```
// * = axi | axi4
function void get_config
(
   input *_config_e config_name,
);
```

Arguments

config_name (AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS
- AXI_CONFIG_MAX_OUTSTANDING_WR
- AXI_CONFIG_MAX_OUTSTANDING_RD

(AXI4) Configuration name:
AXI4_CONFIG_SETUP_TIME
AXI4_CONFIG_HOLD_TIME
AXI4_CONFIG_BURST_TIMEOUT_FACTOR
AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
AXI4_CONFIG_ENABLE_RLAST
AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4_CONFIG_ENABLE_ASSERTION
AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
AXI4_CONFIG_ENABLE_QOS
AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
AXI4_CONFIG_SLAVE_START_ADDR
AXI4_CONFIG_SLAVE_END_ADDR
AXI4_CONFIG_MAX_OUTSTANDING_WR
AXI4_CONFIG_MAX_OUTSTANDING_RD
AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE
AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE

Returns config_val See Slave BFM Configuration for descriptions and valid values.

AXI3 Example

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS);
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR);
```

AXI4 Example

```
get_config(AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE);
get_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_slave_transaction()

This nonblocking function creates a slave transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns with the **_transaction* record.

Prototype

```
// * = axi | axi4
// ** = AXI| AXI4
function automatic *_transaction create_write_transaction();
```

Protocol Transaction Fields

addr	Start address
burst_length	Burst length. Default: 0.
size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;
lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
cache	(AXI3) Burst cache: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;

Protocol Transaction Fields	cache	(AXI4) Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;
	prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
	id	Burst ID.
	data_words	Data words array.
	write_strobes	Write strobes array: Each strobe 0 or 1.
	resp	Burst response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;
	region	(AXI4) Region identifier.
	qos	(AXI4) Quality-of-Service identifier.
	addr_user	Address channel user data.
	data_user	(AXI4) Data channel user data.
	resp_user	(AXI4) Response channel user data.
	read_or_write	Read or write transaction flag: **_TRANS_READ; **_TRANS_WRITE
Operational Transaction Fields	gen_write_strobes	Correction of write strobes for invalid byte lanes: 0 = write_strobes passed through to protocol signals. 1 = write_strobes auto-corrected for invalid byte lanes (default).
	operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING; (default)
	delay_mode	(AXI3) Delay mode: AXI_VALID2READY; (default) AXI_TRANS2READY;

Operational Transaction Fields

write_data_mode	Write data mode: **_DATA_AFTER_ADDRESS; The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS; (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.
address_valid_delay	Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction (default = 0).
data_valid_delay	Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
write_response_ready_delay	Write response channel BREADY delay measured in ACLK cycles for this transaction (default = 0).
data_beat_done	Write data channel beat *done* flag array for this transaction.
transaction_done	Write transaction *done* flag for this transaction.

Returns The *_*transaction* record.

Example

```
// Create a slave transaction.
trans = bfm.create_slave_transaction();
```

execute_read_data_burst()

This task executes a slave read data burst previously created by the *create_slave_transaction()* function. The transaction can be blocking (default) or nonblocking, as defined by the transaction record *operation_mode* field.

It calls the *execute_read_data_phase()* task for each beat of the data burst, with the length of the burst defined by the transaction *burst_length* field.

Prototype

```
// * = axi | axi4
task automatic execute_read_data_burst
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a slave transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_slave_transaction();

....

// Execute the read_trans read data burst.
bfm.execute_read_data_burst(read_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a slave transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_slave_transaction();

....

// Execute the read_trans read data burst.
bfm.execute_read_data_burst(read_trans);
```

execute_read_data_phase()

This task executes a read data phase (beat) previously created by the *create_slave_transaction()* task. This phase can be blocking (default) or nonblocking, as defined by the transaction record *operation_mode* field.

The *execute_read_data_phase()* sets the RVALID protocol signal at the appropriate time defined by the transaction record *data_valid_delay* field and sets the *data_beat_done* array *index* element field to 1 on completion of the phase. If this is the last phase (beat) of the burst, then this task sets the *transaction_done* field to 1 to indicate the whole read transaction has completed.

AXI3 Example

Prototype

```
// * = axi | axi4
task automatic execute_read_data_phase
(
   *_transaction trans,

   int index = 0 // Optional
);
```

Arguments

trans	The *_*transaction* record.
index	Data phase (beat) number.

Returns None

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a slave transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_slave_transaction();

....

// Execute the read data phase for the first beat of the
// read_trans transaction.
bfm.execute_read_data_phase(read_trans, 0);

// Execute the read data phase for the second beat of the
// read_trans transaction.
bfm.execute_read_data_phase(read_trans, 1);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a slave transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_slave_transaction();

....

// Execute the read data phase for the first beat of the
// read_trans transaction.
bfm.execute_read_data_phase(read_trans, 0);

// Execute the read data phase for the second beat of the
// read_trans transaction.
bfm.execute_read_data_phase(read_trans, 1);
```

execute_write_response_phase()

This task executes a write phase previously created by the *create_slave_transaction()* task. This phase can be blocking (default) or nonblocking, as defined by the transaction record *operation_mode* field.

It sets the BVALID protocol signal at the approriate time defined by the transaction record *write_response_valid_delay* field and sets the *transaction_done* field to 1 on completion of the phase to indicate the whole transaction has completed.

Prototype

```
// * = axi | axi4
task automatic execute_write_response_phase
(
   *_transaction trans
);
```

Arguments trans — The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();

....

// Execute the write response phase for the write_trans transaction.
bfm.execute_write_response_phase(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();

....

// Execute the write response phase for the write_trans transaction.
bfm.execute_write_response_phase(write_trans);
```

get_write_addr_phase()

Note

This blocking task gets a write address phase previously created by the *create_slave_transaction()* function. For AXI3, the *get_write_addr_phase()* also sets the AWREADY protocol signal at the appropriate time, defined by the transaction record *address_ready_delay* field.

Prototype

```
// * = axi | axi4
task automatic get_write_addr_phase
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns

None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();

....

// Get the write address phase of the write_trans transaction.
bfm.get_write_addr_phase(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();

....

// Get the write address phase of the write_trans transaction.
bfm.get_write_addr_phase(write_trans);
```

get_read_addr_phase()

This blocking task gets a read address phase previously created by the *create_slave_transaction()* function.

Note

For AXI3, the *get_read_addr_phase()* also sets the ARREADY protocol signal at the appropriate time, defined by the transaction record *address_ready_delay* field.

Prototype

```
// * = axi | axi4
task automatic get_read_addr_phase
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns

None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a slave transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_slave_transaction();

....

// Get the read address phase of the read_trans transaction.
bfm.get_read_addr_phase(read_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a slave transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_slave_transaction();

....

// Get the read address phase of the read_trans transaction.
bfm.get_read_addr_phase(read_trans);
```

get_write_data_phase()

This blocking task gets a write data phase previously created by the *create_slave_transaction()* function. The *get_write_data_phase()* sets the *data_beat_done* array *index* element field to 1 when the phase completes. If this is the last phase (beat) of the burst, then it returns the transaction *last* argument set to 1 to indicate the whole burst is complete.

Note

For AXI3, the *get_write_data_phase()* also sets the WREADY protocol signal at the appropriate time, defined by the transaction record *data_ready_delay* field.

Prototype

```
// * = axi | axi4
task automatic get_write_data_phase
(
   *_transaction trans
   ,
   int index = 0, // Optional
   output bit last
);
```

Arguments

trans	The **_transaction* record.
index	(Optional) Data phase (beat) number.

Returns

last	Flag to indicate that this data phase is the last in the burst.

get_write_data_burst()

This blocking task gets a write data burst previously created by the *create_slave_transaction()* function.

It calls the *get_write_data_phase()* task for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
// * = axi | axi4
task automatic get_write_data_burst
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns

None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();

....

// Get the write data burst of write_trans transaction.
bfm.get_write_data_burst(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a slave transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_slave_transaction();

....

// Get the write data burst of the write_trans transaction.
bfm.get_write_data_burst(write_trans);
```

get_read_addr_cycle()

This blocking AXI4 task waits until the read address channel ARVALID signal is asserted.

Prototype `task automatic get_read_addr_cycle();`
Arguments None
Returns None

AXI3 BFM

Note
The *get_read_addr_cycle()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Waits until the read address channel ARVALID signal is asserted.
bfm.get_read_addr_cycle();
```

execute_read_addr_ready()

This AXI4 task executes a read address ready by placing the *ready* argument value onto the ARREADY signal. It will block for one ACLK period.

Prototype

```
task automatic execute_read_addr_ready
(
   bit ready
);
```

Arguments

ready	The value to be placed onto the ARREADY signal.

Returns None

AXI3 BFM

Note

The *execute_read_addr_ready()* task is not available in the AXI3 BFM. Use the *get_read_addr_phase()* task along with the transaction record *address_ready_delay* field.

AXI4 Example

```
// Assert and deassert the ARREADY signal
forever begin
   bfm.execute_read_addr_ready(1'b0);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
   bfm.wait_on(AXI4_CLOCK_POSEDGE);

   bfm.execute_read_addr_ready(1'b1);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```

get_read_data_ready()

This blocking AXI4 task returns the read data ready value of the RREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_read_data_ready
(
   output bit ready
);
```

Arguments

ready	The value of the RREADY signal.

Returns

ready

AXI3 BFM

Note

The *get_read_data_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the RREADY signal
bfm.get_read_data_ready();
```

get_write_addr_cycle()

This blocking AXI4 task waits until the write address channel AWVALID signal is asserted.

Prototype	`task automatic get_write_addr_cycle();`
Arguments	None
Returns	None

AXI3 BFM

Note

The *get_write_addr_cycle()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Wait for a single write address cycle
bfm.get_write_addr_cycle();
```

execute_write_addr_ready()

This AXI4 task executes a write address ready by placing the *ready* argument value onto the AWREADY signal. It will block for one ACLK period.

Prototype

```
task automatic execute_write_addr_ready
(
   bit ready
);
```

Arguments

ready	The value to be placed onto the AWREADY signal

Returns None

AXI3 BFM

Note

The *execute_write_addr_ready()* task is not available in the AXI3 BFM. Use the *get_write_addr_phase()* task along with the transaction record *address_ready_delay* field.

AXI4 Example

```
// Assert and deassert the AWREADY signal
forever begin
   bfm.execute_write_addr_ready(1'b0);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
   bfm.wait_on(AXI4_CLOCK_POSEDGE);

   bfm.execute_write_addr_ready(1'b1);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```

get_write_data_cycle()

This blocking AXI4 task waits for a single write data cycle for which the WVALID signal is asserted. It will block for one ACLK period.

Prototype `task automatic get_write_data_cycle();`
Arguments None
Returns None

AXI3 BFM

Note
The *get_write_data_cycle()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Wait for a single write data cycle
bfm.get_write_data_cycle();
```

execute_write_data_ready()

This AXI4 task executes a write data ready by placing the *ready* argument value onto the WREADY signal. It will block for one ACLK period.

Prototype

```
task automatic execute_write_data_ready
(
   bit ready
);
```

Arguments

ready	The value to be placed onto the WREADY signal

Returns None

AXI3 BFM

Note

The *execute_write_data_ready()* task is not available in the AXI3 BFM. Use the *get_write_data_phase()* task along with the transaction record *data_ready_delay* field.

AXI4 Example

```
// Assert and deassert the WREADY signal
forever begin
   bfm.execute_write_data_ready(1'b0);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
   bfm.wait_on(AXI4_CLOCK_POSEDGE);

   bfm.execute_write_data_ready(1'b1);

   bfm.wait_on(AXI4_CLOCK_POSEDGE);
end
```

get_write_resp_ready()

This blocking AXI4 task returns the write response ready value of the BREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_write_resp_ready
(
   output bit ready
);
```

Arguments

ready	The value of the BREADY signal.

Returns

readyt

AXI3 BFM

Note

The *get_write_resp_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the BREADY signal
bfm.get_write_resp_ready();
```

wait_on()

This blocking task waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
// * = axi | axi4
// ** = AXI| AXI4
task automatic wait_on
(
   *_wait_e phase,
   input int count = 1 //Optional
);
```

Arguments

phase	Wait for: **_CLOCK_POSEDGE **_CLOCK_NEGEDGE **_CLOCK_ANYEDGE **_CLOCK_0_TO_1 **_CLOCK_1_TO_0 **_RESET_POSEDGE **_RESET_NEGEDGE **_RESET_ANYEDGE **_RESET_0_TO_1 **_RESET_1_TO_0
count	(Optional) Wait for a number of events to occur set by *count*. (default = 1)

Returns None

AXI3 Example

```
bfm.wait_on(AXI_RESET_POSEDGE);
bfm.wait_on(AXI_CLOCK_POSEDGE,10);
```

AXI4 Example

```
bfm.wait_on(AXI4_RESET_POSEDGE);
bfm.wait_on(AXI4_CLOCK_POSEDGE,10);
```

Helper Functions

AMBA AXI protocols typically provide a start address only in a transaction, with the following addresses for each byte of a data burst calculated using the size, length, and type transaction fields of the transaction. Helper functions provide you with a simple interface to set and get address/data values.

get_write_addr_data()

This nonblocking function returns the actual address *addr* and *data* of a particular byte in a write data burst. It also returns the maximum number of bytes (*dynamic_size*) in the write data phase (beat). It is used in a slave test program as a helper function to store a byte of data at a particular address in the slave memory. If the corresponding *index* does not exist, then this function returns *false*; otherwise, it returns *true*.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function bit get_write_addr_data
(
   input *_transaction trans,
   input int index = 0,
   output bit [((**_ADDRESS_WIDTH) - 1): 0] addr[],
   output bit [7:0] data[]
);
```

Arguments

trans	The **_transaction* record.
index	Data words array element number.

Returns

addr	Write address.
data	Write data byte.
bit	Flag to indicate existence of *data* at *index* array element number; 0 = array element nonexistent. 1 = array element exists.

Example

```
bfm.get_write_addr_data(write_trans, 0, addr, data);
```

get_read_addr()

This nonblocking function returns the address *addr* of a particular byte in a read transaction. It is used in a slave test program as a helper function to return the address of a data byte in the slave memory. If the corresponding *index* does not exist, then this function returns *false*; otherwise, it returns *true*.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function bit get_read_addr
(
   input *_transaction trans,
   input int index = 0,
   output bit [((**_ADDRESS_WIDTH) - 1) : 0] addr[]
);
```

Arguments

trans	The **_transaction* record.
index	Array element number.
addr	Read address array

Returns

bit	Flag to indicate existence of data at *index* array element number; 0 = nonexistent. 1 = exists.

Example

```
bfm.get_read_addr(read_trans, 0, addr);
```

set_read_data()

This nonblocking function sets a read data in the **_transaction* record *data_words* field. It is used in a slave test program as a helper function to read from the slave memory given the address *addr*, data beat *index*, and the read *data* arguments.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function bit set_read_data
(
   input *_transaction trans,
   input int index = 0,
   input bit [((**_ADDRESS_WIDTH) - 1) : 0] addr[],
   input bit [7:0] data[]
);
```

Arguments

trans	The **_transaction* record.
index	(Optional) Data byte array element number.
addr	Read address.
data	Read data byte.

Returns None

Example

```
bfm.set_read_data(read_trans, 0, addr, data);
```

Chapter 5
SystemVerilog AXI3 and AXI4 Monitor BFMs

This chapter describes the SystemVerilog AXI3 and AXI4 monitor BFMs. Each BFM has an API that contains tasks and functions to configure the BFM and to access the dynamic Transaction Record during the lifetime of a transaction.

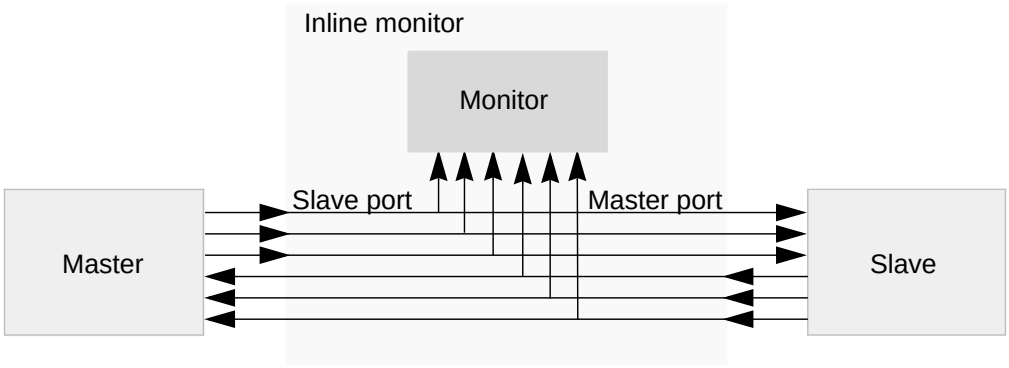
Note

Due to AXI3 protocol specification changes, for some BFM tasks, you reference the AXI3 interface by specifying AXI instead of AXI3.

Inline Monitor Connection

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped in an inline monitor interface and connected inline between a master and slave, as shown in Figure 5-1. It has separate master and slave ports and monitors protocol traffic between a master and slave. The monitor itself then has access to all the facilities provided by the monitor BFM.

Figure 5-1. Inline Monitor Connection Diagram



Monitor BFM Protocol Support

The AXI3 monitor BFM supports the AMBA AXI3 protocol with restrictions described in “Protocol Restrictions” on page 19. In addition to the standard protocol, it supports user sideband signals AWUSER and ARUSER.

The AXI4 monitor BFM supports the AMBA AXI4 protocol with restrictions described in “Protocol Restrictions” on page 19.

Monitor Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following monitor BFM API timing and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the monitor BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

The simulator time-step resolves to the smallest of all the time-precision declarations in the test bench and design IP as a result of these directives, declarations, options, or initialization files:

- `timescale directives in design elements
- Timeprecision declarations in design elements
- Compiler command-line options
- Simulation command-line options
- Local or site-wide simulator initialization files

If there is no timescale directive, the default time unit and time precision are tool specific. The recommended practice is to use timeunit and timeprecision declarations. Refer to the *IEEE Standard for SystemVerilog*, Section 3.14 for details.

Monitor BFM Configuration

The monitor BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters you can use to configure the widths of the address, ID and data signals, and transaction fields to configure timeout factors, slave exclusive support, setup and hold times, and so on.

You can change the address, ID and data signals widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the monitor BFM via a parameter port list of the monitor BFM module. For example, the code extract below shows the AXI3 monitor BFM with the address, ID and data signal widths defined in *module top()* and passed in to the *monitor_test_program* parameter port list:

```
module top ();

   parameter AXI_ADDRESS_WIDTH = 24;
   parameter AXI_RDATA_WIDTH = 16;
   parameter AXI_WDATA_WIDTH = 16;
   parameter AXI_ID_WIDTH = 4;

   monitor_test_program #(AXI_ADDRESS_WIDTH, AXI_RDATA_WIDTH,
AXI_WDATA_WIDTH, AXI_ID_WIDTH) bfm_monitor(....);
```

Table 5-1 lists the parameter names for the address, ID and data signals, and their default values.

Table 5-1. AXI Monitor BFM Signal Width Parameters

Signal Width Parameter	Description
**_ADDRESS_WIDTH	Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
**_RDATA_WIDTH	Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_WDATA_WIDTH	Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_ID_WIDTH	ID signal width in bits. This applies to the RID and WID signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
AXI4_USER_WIDTH	(AXI4) User data signal width in bits. This applies to the ARUSER, AWUSER, RUSER, WUSER and BUSER signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 8.
AXI4_REGION_MAP_SIZE	(AXI4) Region signal width in bits. This applies to the ARREGION and AWREGION signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 16.
index	Ignored for the SystemVerilog monitor BFM.
READ_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding read transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
WRITE_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding write transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
COMBINED_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding combined read and write transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.

Table 5-1. AXI Monitor BFM Signal Width Parameters (cont.)

USE_*	(AXI4) Each protocol signal connection to the monitor BFM can be enabled or disabled. This parameter is set with the Qsys Parameter Editor. See "Running the Qsys Tool" on page 676. for details. 0 = disabled. 1 = enabled (default).

A monitor BFM has configuration fields that you can set via the *set_config()* function to configure variables such as timeout factors, slave exclusive support, and setup and hold times. You can also get the value of a configuration field via the *get_config()* function. Table 5-2 describes the full list of configuration fields.

Table 5-2. AXI Monitor BFM Configuration

Configuration Field	Description
Timing Variables	
**_CONFIG_SETUP_TIME	The setup time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_HOLD_TIME	The hold time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_MAX_TRANSACTION_ TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
**_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
**_CONFIG_MAX_LATENCY_ AWVALID_ASSERTION_TO_AWREADY	The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_ARVALID_ ASSERTION_TO_ARREADY	The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_RVALID_ ASSERTION_TO_RREADY	The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods. Default: 10000.

Table 5-2. AXI Monitor BFM Configuration (cont.)

Configuration Field	Description
**_CONFIG_MAX_LATENCY_BVALID_ ASSERTION_TO_BREADY	The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_WVALID_ ASSERTION_TO_WREADY	The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods. Default: 10000.
AXI_CONFIG_MASTER_DEFAULT_UNDER_ RESET	(AXI3) The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true
AXI4_CONFIG_ENABLE_QOS	(AXI4) The master participates in the Quality-of-Service (qos) scheme. If a master does not participate, the AWQOS/ARQOS value used in write/read transactions must be b0000.
Master Attributes	
AXI4_CONFIG_ENABLE_RLAST	(AXI4) Configures the support for the optional RLAST signal. 0 = disabled 1 = enabled (default)
Slave Attributes	
**_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)

Table 5-2. AXI Monitor BFM Configuration (cont.)

Configuration Field	Description
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET	(AXI3) The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
**_CONFIG_SLAVE_START_ADDR	Configures the start address map for the slave.
**_CONFIG_SLAVE_END_ADDR	Configures the end address map for the slave.
**_CONFIG_READ_DATA_REORDERING_DEPTH	The slave read reordering depth. Refer to the AMBA AXI Protocol Specification for more details. Default: 1.
Error Detection	
**_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
**_CONFIG_ENABLE_ASSERTION	Individual enable/disable of assertion check in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to Monitor Timing and Events for details of simulator time-steps.

Monitor Assertions

Each monitor BFM performs protocol error checking using built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

AXI3 Assertion Configuration

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI_CONFIG_ENABLE_ASSERTION);

// Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

Note

Do not confuse the AXI_CONFIG_ENABLE_ASSERTION bit vector with the AXI_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI3 assertions, refer to “AXI3 Assertions” on page 725.

AXI4 Assertion Configuration

By default, all built-in assertions are enabled in the monitor AXI4 BFM. To globally disable them in the monitor BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS,0)
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
// Define a local bit vector to hold the value of the assertion bit vector
bit [255:0] config_assert_bitvector;

// Get the current value of the assertion bit vector
config_assert_bitvector = bfm.get_config(AXI4_CONFIG_ENABLE_ASSERTION);

// Assign the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector[AXI4_LOCK_CHANGED_BEFORE_AWREADY] = 0;

// Set the new value of the assertion bit vector
bfm.set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector);
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of AXI4 assertions, refer to "AXI4 Assertions" on page 738.

SystemVerilog Monitor API

This section describes the SystemVerilog Monitor API.

set_config()

This function sets the configuration of the monitor BFM.

Prototype

```
// * = axi | axi4
function void set_config
(
   input *_config_e config_name,
   input *_max_bits_t config_val
);
```

Arguments

config_name — **(AXI3) Configuration name:**

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

(AXI4) Configuration name:
AXI4_CONFIG_SETUP_TIME
AXI4_CONFIG_HOLD_TIME
AXI4_CONFIG_BURST_TIMEOUT_FACTOR
AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
AXI4_CONFIG_ENABLE_RLAST
AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4_CONFIG_ENABLE_ASSERTION
AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
AXI4_CONFIG_ENABLE_QOS
AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
AXI4_CONFIG_SLAVE_START_ADDR
AXI4_CONFIG_SLAVE_END_ADDR

config_val — See “Monitor BFM Configuration” on page 124 for descriptions and valid values.

Returns None

AXI3 Example

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1);
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

AXI4 Example

```
set_config(AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE, 1);
set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000);
```

get_config()

This function gets the configuration of the monitor BFM.

Prototype

```
// * = axi | axi4
function void get_config
(
   input *_config_e config_name,
);
```

Arguments

config_name

(AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

(AXI4) Configuration name:

- AXI4_CONFIG_SETUP_TIME
- AXI4_CONFIG_HOLD_TIME
- AXI4_CONFIG_BURST_TIMEOUT_FACTOR
- AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI4_CONFIG_ENABLE_RLAST
- AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
- AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI4_CONFIG_ENABLE_ASSERTION
- AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI4_CONFIG_ENABLE_QOS
- AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI4_CONFIG_SLAVE_START_ADDR
- AXI4_CONFIG_SLAVE_END_ADDR

Returns

config_val

See “Monitor BFM Configuration” on page 124 for descriptions and valid values.

AXI3 Example

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS);
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR);
```

AXI4 Example

```
get_config(AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE);
get_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR);
```

create_monitor_transaction()

This nonblocking function creates a monitor transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns with the **_transaction* record.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function automatic *_transaction create_monitor_transaction();
```

Protocol Transaction Fields

addr	Start address
burst_length	(Optional) Burst length. Default: 0.
size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;
lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
cache	(AXI3) Burst cache: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW;

Protocol Transaction Fields	cache	(AXI4)Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;
	prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
	id	Burst ID
	data_words	Data words array.
	write_strobes	Write strobes array: Each strobe 0 or 1.
	resp	Burst response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;
	region	(AXI4) Region identifier.
	qos	(AXI4) Quality-of-Service identifier.
	addr_user	Address channel user data.
	data_user	(AXI4) Data channel user data.
	resp_user	(AXI4) Response channel user data.
Operational Transaction Fields	gen_write_strobes	Generate write strobes flag: 0 = user supplied write strobes. 1 = auto-generated write strobes (default).
	operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING; (default)
	delay_mode	(AXI3) Delay mode: AXI_VALID2READY; (default) AXI_TRANS2READY;

	write_data_mode	Write data mode: **_DATA_AFTER_ADDRESS; The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS; (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.
Operational Transaction Fields	address_valid_delay	Address channel AWVALID delay measured in ACLK cycles for this transaction (default = 0).
	data_valid_delay	Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
	write_response_ready_delay	Write response channel BREADY delay measured in ACLK cycles for this transaction (default = 0).
	data_beat_done	Write data channel beat *done* flag array for this transaction.
	transaction_done	Write transaction *done* flag for this transaction.
Returns	The *_*transaction* record	

Example

```
// Create a monitor transaction
trans = bfm.create_monitor_transaction();
```

get_rw_transaction()

This blocking task gets a complete read or write transaction previously created by the *create_monitor_transaction()* function.

It updates the **_transaction* record for the complete transaction.

Prototype

```
// * = axi | axi4
task automatic get_rw_transaction
(
   *_transaction trans
)
```

Arguments

trans	The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction monitor_trans;

// Create a monitor transaction and assign it to the local
// monitor_trans variable.
monitor_trans = bfm.create_monitor_transaction();

....

// Get the complete monitor_trans transaction.
bfm.get_rw_transaction(monitor_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction monitor_trans;

// Create a monitor transaction and assign it to the local
// monitor_trans variable.
monitor_trans = bfm.create_monitor_transaction();

....

// Get the complete monitor_trans transaction.
bfm.get_rw_transaction(monitor_trans);
```

get_write_addr_phase()

This blocking task gets a write address phase previously created by the *create_monitor_transaction()* function.

Prototype

```
// * = axi | axi4
task automatic get_write_addr_phase
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write address phase of the write_trans transaction.
bfm.get_write_addr_phase(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write address phase of the write_trans transaction.
bfm.get_write_addr_phase(write_trans);
```

get_read_addr_phase()

This blocking task gets a read address phase previously created by the *create_monitor_transaction()* function.

Prototype

```
// * = axi | axi4
task automatic get_read_addr_phase
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Get the read address phase of the read_trans transaction.
bfm.get_read_addr_phase(read_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Get the read address phase of the read_trans transaction.
bfm.get_read_addr_phase(read_trans);
```

get_read_data_phase()

This blocking task gets a read data phase previously created by the *create_monitor_transaction()* function. The *get_read_data_phase()* sets the *data_beat_done* array *index* element field to 1 when the phase completes. If this is the last phase (beat) of the burst, then it sets the *transaction_done* field to 1 to indicate the whole read transaction is complete.

Prototype

```
// * = axi | axi4
task automatic get_read_data_phase
(
   *_transaction trans
   ,
   int index = 0 // Optional
);
```

Arguments

trans	The **_transaction* record.
index	(Optional) Data phase (beat) number.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Get the read data phase for the first beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 0);

// Get the read data phase for the second beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 1);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Execute the read data phase for the first beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 0);

// Get the read data phase for the second beat of the
// read_trans transaction.
bfm.get_read_data_phase(read_trans, 1);
```

get_read_data_burst()

This blocking task gets a read data burst previously created by the *create_monitor_transaction()* function.

It calls the *get_read_addr_ready()* task for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
// * = axi | axi4
task automatic get_read_data_burst
(
   *_transaction trans
);
```

Arguments

trans	The **_transaction* record.

Returns

None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Get the read data burst of read_trans transaction.
bfm.get_read_data_burst(read_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction read_trans;

// Create a monitor transaction and assign it to the local
// read_trans variable.
read_trans = bfm.create_monitor_transaction();

....

// Get the read data burst of the read_trans transaction.
bfm.get_read_data_burst(read_trans);
```

get_write_data_phase()

This blocking task gets a write data phase previously created by the *create_monitor_transaction()* function. The *get_write_data_phase()* sets the *data_beat_done* array *index* element field to 1 when the phase completes. If this is the last phase (beat) of the burst, then it returns the transaction *last* argument set to 1 to indicate the whole burst is complete.

Prototype

```
// * = axi | axi4
task automatic get_write_data_phase
(
   *_transaction trans
   ,
   int index = 0, // Optional
   output bit last
);
```

Arguments

trans	The **_transaction* record.
index	(Optional) Data phase (beat) number.

Returns

last	Flag to indicate that this data phase is the last in the burst.

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write data phase for the first beat of the
// write_trans transaction.
bfm.get_write_data_phase(write_trans, 0, last);

// Get the write data phase for the second beat of the
// write_trans transaction.
bfm.get_write_data_phase(write_trans, 1, last);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Execute the write data phase for the first beat of the
// write_trans transaction.
bfm.get_write_data_phase(write_trans, 0, last);

// Get the write data phase for the second beat of the
// write_trans transaction.
bfm.get_write_data_phase(write_trans, 1, last);
```

get_write_data_burst()

This blocking task gets a write data burst previously created by the *create_monitor_transaction()* function.

It calls the *get_read_data_phase()* task for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
// * = axi | axi4
task automatic get_write_data_burst
(
   *_transaction trans
);
```

Arguments trans — The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write data burst of write_trans transaction.
bfm.get_write_data_burst(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write data burst of the write_trans transaction.
bfm.get_write_data_burst(write_trans);
```

get_write_response_phase

This blocking task gets a write response phase previously created by the *create_monitor_transaction()* task.

It sets the *transaction_done* field to 1 when the transaction completes to indicate the whole transaction is complete

Prototype

```
// * = axi | axi4
task automatic get_write_response_phase
(
   *_transaction trans
);
```

Arguments trans — The **_transaction* record.

Returns None

AXI3 Example

```
// Declare a local variable to hold the transaction record.
axi_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write response phase of the write_trans transaction.
bfm.get_write_response_phase(write_trans);
```

AXI4 Example

```
// Declare a local variable to hold the transaction record.
axi4_transaction write_trans;

// Create a monitor transaction and assign it to the local
// write_trans variable.
write_trans = bfm.create_monitor_transaction();

....

// Get the write response phase of the write_trans transaction.
bfm.get_write_response_phase(write_trans);
```

get_read_addr_ready()

This blocking AXI4 task returns the read address ready value of the ARREADY signal using the ready argument. It will block for one ACLK period.

Prototype

```
task automatic get_read_addr_ready
(
   output bit ready
);
```

Arguments

ready	The value of the ARREADY signal.

Returns None

AXI3 BFM

Note

The *get_read_addr_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the ARREADY signal value
bfm.get_read_addr_ready();
```

get_read_data_ready()

This blocking AXI4 task returns the read data ready value of the RREADY signal using the ready argument. It will block for one ACLK period.

Prototype

```
task automatic get_read_data_ready
(
   output bit ready
);
```

Arguments

ready	The value of the RREADY signal.

Returns None

AXI3 BFM

Note

The *get_read_data_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the RREADY signal
bfm.get_read_data_ready();
```

get_write_addr_ready()

This blocking AXI4 task returns the write address ready value of the AWREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_write_addr_ready
(
   output bit ready
);
```

Arguments

ready	The value of the AWREADY signal.

Returns None

AXI3 BFM

Note

The *get_write_addr_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the AWREADY signal
bfm.get_write_addr_ready();
```

get_write_data_ready()

This blockingAXI4 task returns the write data ready value of the WREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_write_data_ready
(
   output bit ready
);
```

Arguments

ready	The value of the WREADY signal.

Returns None

AXI3 BFM

Note

The *get_write_data_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the WREADY signalbfm.
get_write_data_ready();
```

get_write_resp_ready()

This blocking AXI4 task returns the write response ready value of the BREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
task automatic get_write_resp_ready
(
   output bit ready
);
```

Arguments

ready	The value of the BREADY signal.

Returns None

AXI3 BFM

Note

The *get_write_resp_ready()* task is not available in the AXI3 BFM.

AXI4 Example

```
// Get the value of the BREADY signal
bfm.get_write_resp_ready();
```

wait_on()

This blocking task waits for an event(s) on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*

Prototype

```
// * = axi | axi4
// ** = AXI| AXI4
task automatic wait_on
(
   *_wait_e phase,
   input int count = 1 //Optional
);
```

Arguments

phase	Wait for: **_CLOCK_POSEDGE **_CLOCK_NEGEDGE **_CLOCK_ANYEDGE **_CLOCK_0_TO_1 **_CLOCK_1_TO_0 **_RESET_POSEDGE **_RESET_NEGEDGE **_RESET_ANYEDGE **_RESET_0_TO_1 **_RESET_1_TO_0
count	(Optional) Wait for a number of events to occur set by *count*. (default = 1)

Returns None

AXI3 Example

```
bfm.wait_on(AXI_RESET_POSEDGE);
bfm.wait_on(AXI_CLOCK_POSEDGE,10);
```

AXI4 Example

```
bfm.wait_on(AXI4_RESET_POSEDGE);
bfm.wait_on(AXI4_CLOCK_POSEDGE,10);
```

Helper Functions

AMBA AXI protocols typically provide a start address only in a transaction, with the following addresses for each byte of a data burst calculated using the size, length, and type transaction fields. Helper functions provide you with a simple interface to set and get address/data values.

get_write_addr_data()

This nonblocking function returns the actual address *addr* and *data* of a particular byte in a write data burst. It is used in a monitor test program as a helper function to store a byte of data at a particular address in the monitor memory. If the corresponding *index* does not exist, then this function returns *false*; otherwise, it returns *true*.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function bit get_write_addr_data
(
   input *_transaction trans,
   input int index = 0,
   output bit [((**_ADDRESS_WIDTH) - 1) : 0] addr[],
   output bit [7:0] data[]
);
```

Arguments

trans	The **_transaction* record.
index	Array element number.
addr	Write address array
data	Write data array

Returns

bit	Flag to indicate existence of *index* array element; 0 = array element non-existent. 1 = array element exists.

Example

```
bfm.get_write_addr_data(write_trans, 0, addr, data);
```

get_read_addr()

This nonblocking function returns the actual address *addr* of a particular index in a read transaction. It is used in a monitor test program as a helper function to return the address of a byte of data in the monitor memory. If the corresponding *index* does not exist, then this function returns *false*; otherwise, it returns *true*.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function bit get_read_addr
(
   input *_transaction trans,
   input int index = 0,
   output bit [((**_ADDRESS_WIDTH) - 1) : 0] addr[]
);
```

Arguments

trans	The *_*transaction* record.
index	Array element number.
addr	Read address array

Returns

bit	Flag to indicate existence of *index* array element; 0 = array element non-existent. 1 = array element exists.

Example

```
bfm.get_read_addr(read_trans, 0, addr);
```

set_read_data()

This nonblocking function sets the read data in the **_transaction* record *data_words* field. It is used in a monitor test program as a helper function to read from the monitor memory given the address *addr*, data beat *index*, and the read *data* arguments.

Prototype

```
// * = axi | axi4
// ** = AXI | AXI4
function bit set_read_addr_data
(
   input *_transaction trans,
   input int index = 0,
   input bit [((**_ADDRESS_WIDTH) - 1) : 0] addr[],
   input bit [7:0] data[]
);
```

Arguments

trans	The **_transaction* record.
index	(Optional) Array element number.
addr	Read address array
data	Read data array

Returns None

Example

```
bfm.set_read_data(read_trans, 0, addr, data);
```

Chapter 6
SystemVerilog Tutorials

This chapter discusses how to use the Mentor VIP – Intel FPGA Edition master and slave BFMs to verify slave and master DUT components.

In the Verifying a Slave DUT tutorial, the slave is an on-chip RAM model that is verified using a master BFM and test program. In the Verifying a Master DUT tutorial, the master issues simple write and read transactions that are verified using a slave BFM and test program.

Following this top-level discussion of how you verify a master and a slave component using Mentor VIP – Intel FPGA Edition is a brief example of how to run Qsys, the powerful system integration tool in Quartus® Prime software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details on this example, refer to "Getting Started with Qsys and the BFMs" on page 673.

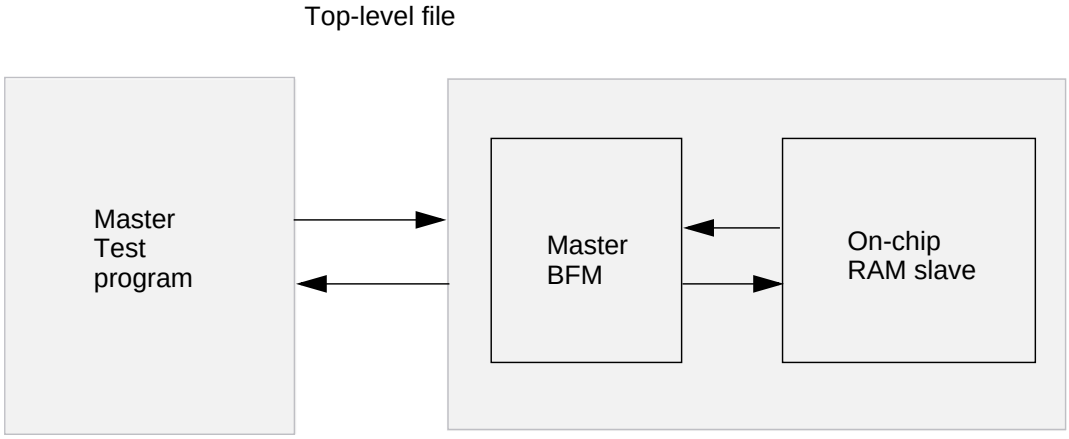
Note

Parameters to configure any optional signals, master BFM transaction issuing and slave BFM acceptance capability, are set with the Qsys Parameter Editor. See "Running the Qsys Tool" on page 676 for details of the Qsys Parameter Editor.

Verifying a Slave DUT

A slave DUT component is connected to a master BFM at the signal-level. A master test program, written at the transaction-level, generates stimulus using the master BFM to verify the slave DUT. Figure 6-1 illustrates a typical top-level test bench environment.

Figure 6-1. Slave DUT Top-Level Test Bench Environment



In this example, the master test program also compares the written data with that read back from the slave DUT, reporting the result of the comparison.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

AXI3 BFM Master Test Programs

Using the AXI3 master BFM API, this master test program creates a wide range of stimulus scenarios that test the slave DUT. This tutorial restricts the stimulus to a write transaction followed by a read transaction to the same address to compare the read data with the previously written data.

Note

For a complete code example of this master test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi3\qsys-examples\ex1_back_to_back_sv\master_test_program.sv

Configuration and Initialization

The code excerpt in Example 6-1 shows the master test program defining nine transaction variables, *trans*, and *trans1* to *trans8*, of type *axi_transaction*, which hold the transaction record for each transaction. A *timeout* transaction field is configured in the AXI3 Master BFM before waiting for the system reset to be completed. An additional system clock cycle is waited on after reset to satisfy the AXI3 protocol requirement specified in Section 11.1.2 of the AMBA AXI Protocol Specification before executing transactions.

Example 6-1. Configuration and Initialization

```
initial
begin
axi_transaction trans trans1, trans2, trans3, trans4;
axi_transaction trans5, trans6, trans7, trans8;

     /******************
    ** Configuration **
    ******************/
    begin
      bfm.set_config(AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR, 1000);
    end

    /*******************
    ** Initialization **
    *******************/
    bfm.wait_on(AXI_RESET_0_TO_1);
    bfm.wait_on(AXI_CLOCK_POSEDGE);
```

Write Transaction Creation and Execution

To generate AXI3 protocol traffic, the master test program must create the transaction *trans* before executing it. The code excerpt in Example 6-6 calls the *create_write_transaction()* function, providing only the start address argument of the transaction. The optional burst-length argument automatically defaults to a value of zero, indicating a burst length of a single beat. Refer to "Master BFM Configuration" on page 52 for more details.

This example has an AXI3 data bus width of 32-bits; therefore, a single beat of data conveys 4 bytes across the data bus. The *set_data_words()* function sets the *data_words[0]* transaction field with the value of 1 on byte lane 1, resulting in a value of 32'h0000_0100. However, the AXI3 protocol permits narrow transfers with the use of the write strobes signal WSTRB to indicate which byte lane contains valid write data, and therefore indicates to the slave DUT which data byte lane will be written into memory. Similarly, you can call the *set_write_strobes()* function to set the *write_strobes[0]* transaction field with the value of 4'b0010, indicating that only valid data is being transferred on byte lane 1. The write transaction, *trans*, then executes on the protocol signals by calling the *execute_transaction()* function.

All other write transaction fields default to legal protocol values. Refer to "Master BFM Configuration" on page 52 for more details.

Example 6-2. Write Transaction Creation and Execution

```
/************************
** Traffic generation: **
************************/
// 4 x Writes
// Write data value 1 on byte lanes 1 to address 1.
trans = bfm.create_write_transaction(1);
trans.set_data_words(32'h0000_0100, 0);
trans.set_write_strobes(4'b0010, 0);
$display ( "@ %t, master_test_program: Writing data (1) to
            address (1)", $time);

// By default it will run in Blocking mode
bfm.execute_transaction(trans);
```

In the complete master test program, three subsequent write transactions are created and executed in a similar manner to Example 6-2.

Read Transaction Creation and Execution

The code excerpt in Example 6-3 reads the data that has been previously written into the slave memory. The master test program first creates a read transaction *trans* by calling the *create_read_transaction()* function, providing only the start address argument. The optional burst-length argument automatically defaults to a value of zero, indicating a burst length of a single beat.

The *set_id()* function is then called to set the transaction *id* field to 1, and the *set_size()* procedure sets the transaction *size* field to be a single byte (AXI_BYTES_1). The read transaction, *trans*, is then executed onto the protocol signals with a call to the *execute_transaction()* function.

The read data is obtained by calling the *get_data_words()* function to get the *data_words[0]* transaction field value. The result of the read data is compared with the expected data, and a message displays the transcript.

Example 6-3. Read Transaction Creation and Execution

```
// Read data from address 1.
trans = bfm.create_read_transaction(1);
trans.set_size(AXI_BYTES_1);
trans.set_id(1);

bfm.execute_transaction(trans);
if (trans.get_data_words(0) == 32'h0000_0100)
   $display ( "@ %t, master_test_program: Read correct data (1) at
              address (1)", $time);
else
   $display ( "@ %t master_test_program: Error: Expected data (1) at
         address 1, but got %d", $time, trans.get_data_words(0));
```

In the complete master test program, three subsequent read transactions are created and executed in a similar manner to Example 6-3.

Write Burst Transaction Creation and Execution

The code excerpt in Example 6-4 calls the *create_write_transaction()* function to create a write burst transaction, *trans*, by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 7+1=8.

Note

The burst length argument passed to the *create_write_transaction()* function is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the AWLEN protocol signals.

The *set_data_words()* function is then called eight times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst; therefore, a *for* loop calls the *set_write_strobes()* function to set the *write_strobes* fields of the transaction to 4’b1111 for each beat of the burst.

The write transaction is then executed onto the protocol signals.

Example 6-4. Write Burst Transaction Creation and Execution

```
// Write data burst length of 7 to start address 16.
trans = bfm.create_write_transaction(16, 7);

trans.set_data_words('hACE0ACE1, 0);
trans.set_data_words('hACE2ACE3, 1);
trans.set_data_words('hACE4ACE5, 2);
trans.set_data_words('hACE6ACE7, 3);
trans.set_data_words('hACE8ACE9, 4);
trans.set_data_words('hACEAACEB, 5);
trans.set_data_words('hACECACED, 6);
trans.set_data_words('hACEEACEF, 7);
for(int i=0; i<8; i++)
   trans.set_write_strobes(4'b1111, i);

$display ( "@ %t, master_test_program: Writing data burst of length 7 to
            start address 16", $time);

bfm.execute_transaction(trans);
```

In the complete master test program, a subsequent write data burst transaction with a start address of 128 is created and executed in similar manner to Example 6-4.

Read Burst Transaction Creation and Execution

The code excerpt in Example 6-5 reads the first two data beats from the data burst that has been previously written into the slave memory. The call to the *create_read_transaction()* function creates the read burst transaction *trans* by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 1+1=2.

Note

The burst length argument passed to the *create_read_transaction()* function is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the ARLEN protocol signals.

The read transaction, *trans*, is then executed onto the protocol signals by calling the *execute_transaction()* function. The read data is obtained by calling the *get_data_words(n)* function to get the *data_words[n]* transaction field value. The result of the read data is compared with the expected data, and a message displays the transcript.

Example 6-5. Read Burst Transaction Creation and Execution

```
// Read data burst of length 1 from address 16.
trans = bfm.create_read_transaction(16, 1);

bfm.execute_transaction(trans);
if (trans.get_data_words(0) == 'hACE0ACE1)
   $display ( "@ %t, master_test_program: Read correct data (hACE0ACE1) at
               address (16)", $time);
else
   $display ( "@ %t, master_test_program: Error: Expected data (hACE0ACE1)
      at address (16), but got %h", $time, trans.get_data_words(0));

if (trans.get_data_words(1) == 'hACE2ACE3)
   $display ( "@ %t, master_test_program: Read correct data (hACE2ACE3) at
            address (20)", $time);
else
   $display ( "@ %t, master_test_program: Error: Expected data (hACE2ACE3)
            at address (20), but got %h", $time, trans.get_data_words(1));
```

In the complete master test program, a subsequent read transaction with a start address of 128 is created and executed in a similar manner to Example 6-3.

Create and Execute Outstanding Write Burst Transactions

The code excerpt in Example 6-6 uses the AXI3 Master BFM *create_write_transaction()* function to create a write burst transaction *trans1* by providing the start address and burst length arguments. The actual length of the burst on the protocol wires is 3+1=4.

Note

The burst length argument passed to the *create_write_transaction()* function is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the AWLEN protocol signals.

The *set_data_words()* function is then called four times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst; therefore, a *for* loop uses the *set_write_strobes()* function to set the *write_strobes* fields of the transaction to 4’b1111.

The write transaction *trans1* is then executed onto the protocol signals by calling the *execute_write_addr_phase()* and *execute_write_data_burst()* functions in parallel within a *fork.. join_any* statement. By calling the *execute_write_addr_phase()* and *execute_write_data_burst()* functions in parallel, subsequent address phase transactions can be executed before the current write data burst has completed. This allows outstanding write transaction stimulus to be created.

Example 6-6. Create and Execute Outstanding Write Burst Transactions

```
/***********************************
** Outstanding Traffic generation: **
***********************************/
// 4 Outstanding Write Transactions
// Write data value to address 0.
trans1 = bfm.create_write_transaction(0,3); //Burst length=3+1
trans1.set_data_words('hACE0ACE1, 0);
trans1.set_data_words('hACE2ACE3, 1);
trans1.set_data_words('hACE4ACE5, 2);
trans1.set_data_words('hACE6ACE7, 3);
for(int i=0; i<4; i++)
   trans1.set_write_strobes(4'b1111, i);
$display ( "@ %t, master_test_program: Writing data (1) to address (0)",
$time);

fork
   bfm.execute_write_addr_phase(trans1);
   bfm.execute_write_data_burst(trans1);
join_any
```

In the complete master test program, subsequent write transaction are created and assigned to unique variables, *trans2*, *trans3*, *trans4*, and so on, allowing multiple write transactions to exist at the same time. The write transactions are then executed in a similar manner to that shown in Example 6-6, resulting in outstanding write transactions.

AXI4 BFM Master Test Program

A master test program using the master BFM API is capable of creating a wide range of stimulus scenarios to verify a slave DUT. However, this tutorial restricts the master BFM stimulus to write transactions followed by read transactions to the same address, and then compares the read data with the previously written data.

Note

For a complete code example of this master test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi4\qsys-examples\ex1_back_to_back_sv\master_test_program.sv

Mentor VIP – Intel FPGA EditionThe master test program contains the following elements:

- A Configuration and Initialization that creates and executes read and write transactions.
- Tasks *handle_write_resp_ready()* and *handle_read_data_ready()* to handle the delay of the write response channel BREADY signal and the read data channel RREADY signals, respectively.
- Variables *m_wr_resp_phase_ready_delay* and *m_rd_data_phase_ready_delay* to set the delay of the BREADY and RREADY signals

- A *master_ready_delay_mode* variable to configure the behavior of the handshake signals *VALID to *READY delay.

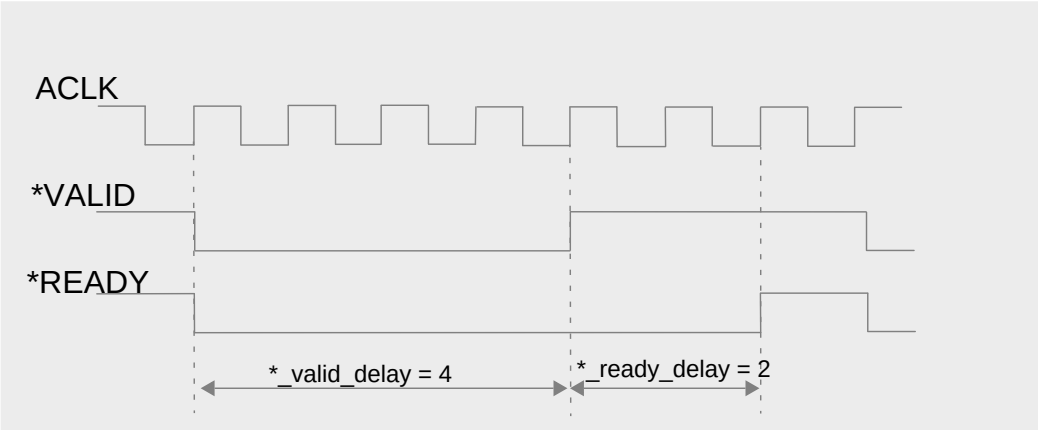
The following sections describe the main tasks and variables.

master_ready_delay_mode

The *master_ready_delay_mode* variable holds the configuration that defines the starting point of any delay applied to the RREADY and BREADY signals. It can be configured to the enumerated type values of AXI4_VALID2READY (default) or AXI4_TRANS2READY.

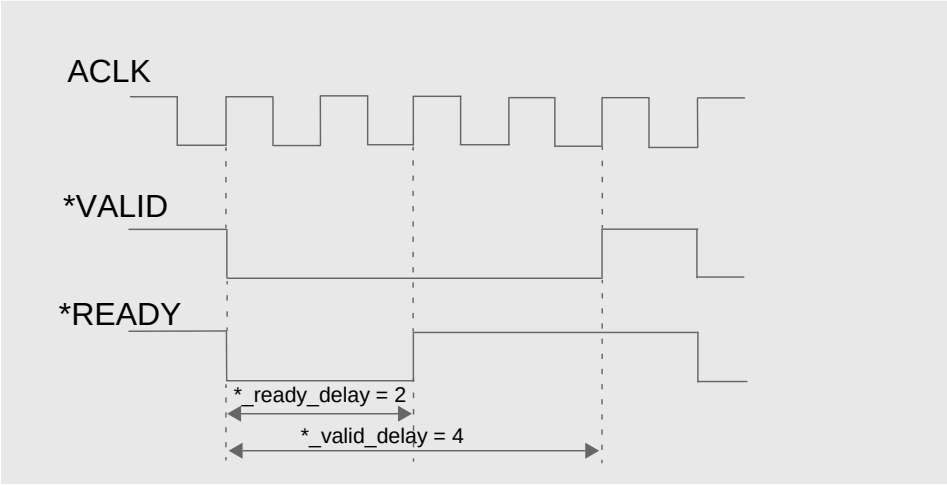
The default configuration (*master_ready_delay_mode* = AXI4_VALID2READY) corresponds to the delay measured from the positive edge of ACLK when *VALID is asserted. Figure 6-2 shows how to achieve a *VALID before *READY handshake, respectively.

Figure 6-2. master_ready_delay_mode = AXI4_VALID2READY



The nondefault configuration (*master_ready_delay_mode* = AXI4_TRANS2READY) corresponds to the delay measured from the completion of a previous transaction phase (*VALID and *READY both asserted). Figure 6-3 shows how to achieve a *READY before *VALID handshake.

Figure 6-3. master_ready_delay_mode = AXI4_TRANS2READY



Example 6-7 shows the configuration of the *master_ready_delay_mode* to its default value.

Example 6-7. master_ready_delay_mode

```
// Enum type for master ready delay mode
// AXI4_VALID2READY - Ready delay for a phase will be applied from
//                 start of phase (Means from when VALID is asserted).
// AXI4_TRANS2READY - Ready delay will be applied from the end of
//                 previous phase. This might result in ready before valid.
typedef enum bit
{
   AXI4_VALID2READY = 1'b0,
   AXI4_TRANS2READY = 1'b1
} axi4_master_ready_delay_mode_e;

// Master ready delay mode selection : default it is VALID2READY
axi4_master_ready_delay_mode_e master_ready_delay_mode =
AXI4_VALID2READY;
```

m_wr_resp_phase_ready_delay

The *m_wr_resp_phase_ready_delay* variable holds the BREADY signal delay. The delay value extends the length of the write response phase by a number of ACLK cycles. The starting point of the delay is determined by the *master_ready_delay_mode* variable configuration.

Example 6-8 shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.

Example 6-8. m_wr_resp_phase_ready_delay

```
// Variable : m_wr_resp_phase_ready_delay
int m_wr_resp_phase_ready_delay = 2;
```

m_rd_data_phase_ready_delay

The *m_rd_data_phase_ready_delay* variable holds the RREADY signal delay. The delay value extends the length of each read data phase (beat) by a number of ACLK cycles. The starting point of the delay is determined by the *master_ready_delay_mode* variable configuration.

Example 6-9 shows the RREADY signal delayed by two ACLK cycles. You can edit this variable to change the RREADY signal delay.

Example 6-9. m_rd_data_phase_ready_delay

```
// Variable : m_rd_data_phase_ready_delay
int m_rd_data_phase_ready_delay = 2;
```

Configuration and Initialization

In an *initial* block, the master test program defines nine transaction variables *trans*, and *trans1* to *trans8*, of type *axi4_transaction*, which hold the record of each transaction during its lifetime, as shown in Example 6-10. The initial wait for the ARESETn signal to be deactivated, followed by a positive ACLK edge, satisfies the protocol requirement detailed in Section A3.1.2 of the AXI Protocol Specification.

Example 6-10. Configuration and Initialization

```
initial
begin
   axi4_transaction trans, trans1, trans2, trans3, trans4;
   axi4_transaction trans5, trans6, trans7, trans8;

   /*******************
   ** Initialization **
   *******************/
   bfm.wait_on(AXI4_RESET_0_TO_1);
   bfm.wait_on(AXI4_CLOCK_POSEDGE);
```

Create and Execute Write Transactions

To generate AXI4 protocol traffic, the master test program must create the transaction *trans* before executing it. The code excerpt in Example 6-11 calls the *create_write_transaction()* function, providing only the start address argument of the transaction. The burst-length argument automatically defaults to a value of zero, indicating a burst length of a single beat.

This example has an AXI4 data bus width of 32 bits; therefore, a single beat of data conveys 4 bytes across the data bus. The call to the *set_data_words()* function sets the *data_words[0]* transaction field with the value of 1 on byte lane 1, resulting in a value of 32'h0000_0100. However, the AXI4 protocol permits narrow transfers with the use of the write strobes signal WSTRB to indicate which byte lane contains valid write data, and therefore indicates to the slave DUT which data byte lane will be written into memory. Similarly, you can call the

set_write_strobes() function to set the *write_strobes[0]* transaction field with the value of 4'b0010, indicating that only valid data is being transferred on byte lane 1. The write transaction *trans* then executes on the protocol signals by calling the *execute_transaction()* function.

All other write transaction fields default to legal protocol values (see *create_write_transaction()* for details).

Example 6-11. Create and Execute Write Transactions

```
/************************
** Traffic generation: **
************************/
// 4 x Writes
// Write data value 1 on byte lanes 1 to address 1.
trans = bfm.create_write_transaction(1);
trans.set_data_words(32'h0000_0100, 0);
trans.set_write_strobes(4'b0010, 0);
$display ( "@ %t, master_test_program: Writing data (1) to address (1)",
$time);

// By default it will run in Blocking mode
bfm.execute_transaction(trans);
```

In the complete master test program, three subsequent write transactions are created and executed in a similar manner to Example 6-11.

Create and Execute Read Transactions

The code excerpt in Example 6-12 reads the data that has been previously written into the slave memory. The master test program first creates a read transaction *trans* by calling the *create_read_transaction()* function, providing only the start address argument. The burst-length argument automatically defaults to a value of zero, indicating a burst length of a single beat. Refer to “create_read_transaction()” on page 65 for more details.

The *set_id()* function is then called to set the transaction *id* field to be 1 before executing the read transaction *trans* onto the protocol signals with a call to the *execute_transaction()* function.

The read data is obtained by calling the *get_data_words(0)* function to get the *data_words[0]* transaction field value. The result of the read data is compared with the expected data, and a message displays the transcript.

Example 6-12. Create and Execute Read Transactions

```
// Read data from address 1.
trans = bfm.create_read_transaction(1);
trans.set_size(AXI4_BYTES_1);
trans.set_id(1);

bfm.execute_transaction(trans);
if (trans.get_data_words(0) == 32'h0000_0100)
   $display ( "@ %t, master_test_program: Read correct data (1) at
             address (1)", $time);
else
   $display ( "@ %t master_test_program: Error: Expected data (1) at
         address 1, but got %d", $time, trans.get_data_words(0));
```

In the complete master test program, three subsequent read transactions are created and executed in a similar manner to Example 6-12.

Create and Execute Write Burst Transactions

The code excerpt in Example 6-13 calls the *create_write_transaction()* function to create a write burst transaction *trans* by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 7+1=8.

Note

The burst length argument passed to the *create_write_transaction()* function is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the AWLEN protocol signals.

The *set_data_words()* function is then called eight times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst; therefore, a *for* loop calls the *set_write_strobes()* function to set the *write_strobes* fields of the transaction to 4’b1111 for each beat of the burst.

The write transaction is then executed onto the protocol signals.

Example 6-13. Create and Execute Write Burst Transactions

```
// Write data burst length of 7 to start address 16.
trans = bfm.create_write_transaction(16, 7);

trans.set_size(AXI4_BYTES_4);
trans.set_data_words('hACE0ACE1, 0);
trans.set_data_words('hACE2ACE3, 1);
trans.set_data_words('hACE4ACE5, 2);
trans.set_data_words('hACE6ACE7, 3);
trans.set_data_words('hACE8ACE9, 4);
trans.set_data_words('hACEAACEB, 5);
trans.set_data_words('hACECACED, 6);
trans.set_data_words('hACEEACEF, 7);
for(int i=0; i<8; i++)
```

```
   trans.set_write_strobes(4'b1111, i);

$display ( "@ %t, master_test_program: Writing data burst of length 7 to
            start address 16", $time);

bfm.execute_transaction(trans);
```

In the complete master test program, a subsequent write data burst transaction with a start address of 128 is created and executed in a similar manner to Example 6-13.

Read Burst Transaction Creation and Execution

The code excerpt in Example 6-14 reads the first two data beats from the data burst that has been previously written into the slave memory. The call to *create_read_transaction()* function creates the read burst transaction *trans* by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 1+1=2.

Note

The burst length argument passed to the *create_read_transaction()* function is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the ARLEN protocol signals.

The read transaction *trans* is then executed onto the protocol signals by calling the *execute_transaction()* function. The read data is obtained by calling the *get_data_words(n)* function to get the *data_words[n]* transaction field value. The result of the read data is compared with the expected data, and a message displays the transcript.

Example 6-14. Read Burst Transaction Creation and Execution

```
// Read data burst of length 1 from address 16.
trans = bfm.create_read_transaction(16, 1);

bfm.execute_transaction(trans);
if (trans.get_data_words(0) == 'hACE0ACE1)
   $display ( "@ %t, master_test_program: Read correct data (hACE0ACE1) at
               address (16)", $time);
else
   $display ( "@ %t, master_test_program: Error: Expected data (hACE0ACE1)
      at address (16), but got %h", $time, trans.get_data_words(0));

if (trans.get_data_words(1) == 'hACE2ACE3)
   $display ( "@ %t, master_test_program: Read correct data (hACE2ACE3) at
            address (20)", $time);
else
   $display ( "@ %t, master_test_program: Error: Expected data (hACE2ACE3)
            at address (20), but got %h", $time, trans.get_data_words(1));
```

In the complete master test program, a subsequent read transaction with a start address of 128 is created and executed in a similar manner to Example 6-14.

Outstanding Write Burst Transaction Creation and Execution

The code excerpt in Example 6-6 calls the AXI4Master BFM *create_write_transaction()* function to create a write burst transaction *trans1* by providing the start address and burst length arguments. The actual length of the burst on the protocol wires is 3+1=4.

The *set_data_words()* function is then called four times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst. Calling the *set_write_strobes()* function sets the *write_strobes* fields of the transaction to 4'b1111.

The write transaction, *trans1*, is then executed onto the protocol signals by calling the *execute_write_addr_phase()* and *execute_write_data_burst()* functions in parallel within a *fork.. join_any* statement. By calling the *execute_write_addr_phase()* and *execute_write_data_burst()* functions in parallel, subsequent address phase transactions can be transmitted before the write data burst has completed.

Example 6-15. Outstanding Write Burst Transaction Creation and Execution

```
/***********************************
** Outstanding Traffic generation: **
***********************************/
// 4 x Writes
// Write data value to address 0.
trans1 = bfm.create_write_transaction(0,3); //Burst length=3+1
trans1.set_data_words('hACE0ACE1, 0);
trans1.set_data_words('hACE2ACE3, 1);
trans1.set_data_words('hACE4ACE5, 2);
trans1.set_data_words('hACE6ACE7, 3);
for(int i=0; i<4; i++)
   trans1.set_write_strobes(4'b1111, i);
$display ( "@ %t, master_test_program: Writing data (1) to address (1)",
$time);

fork
   bfm.execute_write_addr_phase(trans1);
   bfm.execute_write_data_burst(trans1);
join_any
```

In the complete master test program, subsequent write transactions are created and assigned to unique variables, *trans2*, *trans3*, *trans4*, and so on, allowing multiple write transactions to exist at the same time. The write transactions are then executed in a similar manner to that shown in Example 6-6, resulting in outstanding write transactions.

handle_write_resp_ready()

The *handle_write_resp_ready()* task handles the BREADY signal for the write response channel. In a *forever* loop, it delays the assertion of the BREADY signal based on the settings of the *master_ready_delay_mode* and *m_wr_resp_phase_ready_delay* as shown in Example 6-16.

If the *master_delay_ready_mode = AXI4_VALID2READY*, then the BREADY signal is immediately deasserted using the nonblocking call to the *execute_write_resp_ready()* task and waits for a write channel response phase to occur with a call to the blocking *get_write_response_cycle()* task. A received write response phase indicates that the BVALID signal has been asserted, triggering the starting point for the delay of the BREADY signal by the number of ACLK cycles defined by *m_wr_resp_phase_ready_delay*. After the delay, another call to the *execute_write_resp_ready()* task to assert the BREADY signal completes the BREADY handling. The *seen_valid_ready* flag is set to indicate the end of a response phase when both BVALID and BREADY are asserted, and the completion of the write transaction.

If the *master_delay_ready_mode = AXI4_TRANS2READY*, then a check of the *seen_valid_ready* flag is performed to indicate that a previous write transaction has completed. If a write transaction is still active (indicated by either BVALID or BREADY not asserted), then the code waits until the previous write transaction has completed. The BREADY signal is deasserted using the nonblocking call to the *execute_write_resp_ready()* task and waits for the number of ACLK cycles defined by *m_wr_resp_phase_ready_delay*. A nonblocking call to the *execute_write_resp_ready()* task to assert the BREADY signal completes the BREADY handling. The *seen_valid_ready* flag is cleared to indicate that only BREADY has been asserted.

Example 6-16. handle_write_resp_ready()

```
// Task : handle_write_resp_ready
// This method assert/de-assert the write response channel ready signal.
// Assertion and de-assertion is done based on following variable's value:
// m_wr_resp_phase_ready_delay
// master_ready_delay_mode
task automatic handle_write_resp_ready;
   bit seen_valid_ready;

   int tmp_ready_delay;
   axi4_master_ready_delay_mode_e tmp_mode;

   forever
   begin
      wait(m_wr_resp_phase_ready_delay > 0);
      tmp_ready_delay = m_wr_resp_phase_ready_delay;
      tmp_mode        = master_ready_delay_mode;

      if (tmp_mode == AXI4_VALID2READY)
      begin
         fork
            bfm.execute_write_resp_ready(1'b0);
         join_none

         bfm.get_write_response_cycle;
         repeat(tmp_ready_delay - 1) bfm.wait_on(AXI4_CLOCK_POSEDGE);

         bfm.execute_write_resp_ready(1'b1);
         seen_valid_ready = 1'b1;
      end
      else  // AXI4_TRANS2READY
      begin
         if (seen_valid_ready == 1'b0)
         begin
            do
               bfm.wait_on(AXI4_CLOCK_POSEDGE);
            while (!((bfm.BVALID === 1'b1) && (bfm.BREADY === 1'b1)));
         end

         fork
            bfm.execute_write_resp_ready(1'b0);
         join_none

         repeat(tmp_ready_delay) bfm.wait_on(AXI4_CLOCK_POSEDGE);

         fork
            bfm.execute_write_resp_ready(1'b1);
         join_none
         seen_valid_ready = 1'b0;
      end
   end
endtask
```

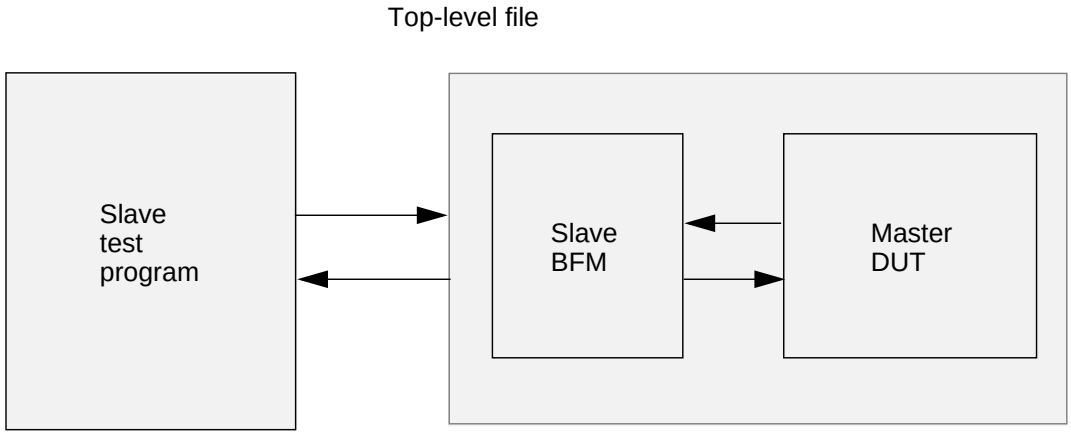
handle_read_data_ready()

The *handle_read_data_ready()* task handles the RREADY signal for the read data channel. It delays the assertion of the RREADY signal based on the settings of *master_ready_delay_mode* and *m_rd_data_phase_ready_delay*. The *handle_read_data_ready()* task code is similar in operation to the *handle_write_resp_ready()* task.

Verifying a Master DUT

A master DUT component is connected to a slave BFM at the signal-level. A slave test program, written at the transaction-level, generates stimulus via the slave BFM to verify the master DUT. Figure 6-4 illustrates a typical top-level test bench environment.

Figure 6-4. Master DUT Top-Level Test Bench Environment



In this example, the slave test program is a simple memory model.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

AXI3 BFM Slave Test Program

The slave test program is a memory model and contains two APIs: a AXI3 Basic Slave API Definition and an Advanced AXI3 Slave API Definition.

The AXI3 Basic Slave API Definition allows you to create a wide range of stimulus scenarios to test a master DUT. This API definition simplifies the creation of slave stimulus based on the default response of OKAY to read and write transactions.

The Advanced AXI3 Slave API Definition allows you to create additional response scenarios to read and write transactions. For example, a successful exclusive transaction requires an EXOKAY response.

Note

For a complete code example of this slave test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi3\qsys-examples\ex1_back_to_back_sv\slave_test_program.sv

AXI3 Basic Slave API Definition

The basic slave test program API contains the following:

- Functions that read and write a byte of data to internal memory *do_byte_read()* and *do_byte_write()*, respectively.
- Functions to configure the AXI3 protocol channel handshake delays *set_read_address_ready_delay()*, *set_write_address_ready_delay()*, *set_write_data_ready_delay()*, *set_read_data_valid_delay()*, and *set_wr_resp_valid_delay()*.
- Tasks to process read and write transactions, *process_read* and *process_write*, respectively. If you need to create other responses, such as EXOKAY, DECERR, or SLVERR, then you will need to edit these tasks to provide the required response.
- A *slave_mode* transaction field controls the behavior of reading and writing to the internal memory.

The *internal memory* for the slave is defined as a sparse array of 8 bits, so that each byte of data is stored as an address/data pair.

Example 6-17. internal memory

```
// Storage for a memory
bit [7:0] mem [*];
```

The *do_byte_read()* function, when called, reads a data byte from the internal memory, *mem*, given an address location as demonstrated in Example 6-18.

You can edit this function to modify the way the read data is extracted from the internal memory.

Example 6-18. do_byte_read()

```
// Function : do_byte_read
// Function to provide read data byte from memory at
// particular input address
function bit[7:0] do_byte_read(addr_t addr);
   return mem[addr];
endfunction
```

The *do_byte_write()* function, when called, writes a data byte to the internal memory, *mem*, given an address location as Example 6-19 illustrates.

You can edit this function to modify the way the write data is stored in the internal memory.

Example 6-19. do_byte_write()

```
// Function : do_byte_write
// Function to write data byte to memory at particular
// input address
function void do_byte_write(addr_t addr, bit [7:0] data);
   mem[addr] = data;
endfunction
```

The *set_read_address_ready_delay()* function, when called, configures the ARREADY handshake signal to be delayed by a number of ACLK cycles, which extends the length of the read address phase. The starting point of the delay is determined by the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details. Example 6-20 demonstrates setting the ARREADY signal delay by four ACLK cycles.

You can edit this function to change the ARREADY signal delay.

Example 6-20. set_read_address_ready_delay()

```
// Function : set_read_address_ready_delay
// This is used to set read address phase ready delay
// to extend phase
function void set_read_address_ready_delay(axi_transaction trans);
   trans.set_address_ready_delay(4);
endfunction
```

The *set_write_address_ready_delay()* function, when called, configures the AWREADY handshake signal to be delayed by a number of ACLK cycles, which extends the length of the write address phase. The starting point of the delay is determined by the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details. Example 6-21 demonstrates setting the AWREADY signal delay by two ACLK cycles.

You can edit this function to change the AWREADY signal delay.

Example 6-21. set_write_address_ready_delay()

```
// Function : set_write_address_ready_delay
// This is used to set write address phase ready delay
// to extend phase
function void set_write_address_ready_delay(axi_transaction trans);
  trans.set_address_ready_delay(2);
endfunction
```

The *set_write_data_ready_delay()* function, when called, configures the WREADY signal handshake to be delayed by a number of ACLK cycles, which extends the length of each write data phase (beat) in a write data burst. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details.

For each write data phase (beat), the delay value of the WREADY signal is stored in an element of the *data_ready_delay[]* array for the transaction, as demonstrated in Example 6-22.

You can edit this function to change the WREADY signal delay.

Example 6-22. set_write_data_ready_delay()

```
// Function : set_write_data_ready_delay
// This will set the ready delays for each write data phase
// in a write data burst
function void set_write_data_ready_delay(axi_transaction trans);
   for (int i = 0; i < trans.data_ready_delay.size(); i++)
      trans.set_data_ready_delay(i, i);
endfunction
```

The *set_read_data_valid_delay()* function, when called, configures the RVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of each read data phase (beat) in a read data burst. The starting point of the delay is determined by the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details.

For each read data phase (beat), the delay value of the RVALID signal is stored in an element of the *data_valid_delay[]* array for the transaction, as demonstrated in Example 6-23.

You can edit this function to change the RVALID signal delay.

Example 6-23. set_read_data_valid_delay()

```
// Function : set_read_data_valid_delay
// This is used to set read response phase valid delays to start
// driving read data/response phases after specified delay.
function void set_read_data_valid_delay(axi_transaction trans);
  for (int i = 0; i < trans.data_valid_delay.size(); i++)
    trans.set_data_valid_delay(i, i);
endfunction
```

The *set_wr_resp_valid_delay()* function, when called, configures the BREADY signal handshake to be delayed by a number of ACLK cycles, which extends the length of the write response phase. The starting point of the delay is determined by the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details. Example 6-24 below demonstrates setting the BREADY signal delay by two ACLK cycles. You can edit this function to change the BREADY signal delay.

Example 6-24. set_wr_resp_valid_delay()

```
// Function : set_wr_resp_valid_delay
// This is used to set write response phase valid delay to start
// driving write response phase after specified delay.
function void set_wr_resp_valid_delay(axi_transaction trans);
   trans.set_write_response_valid_delay(2);
endfunction
```

There is a *slave_mode* transaction field that you can configure to control the behavior of reading and writing to the internal memory. It has two modes AXI_TRANSACTION_SLAVE and AXI_PHASE_SLAVE.

Example 6-25. slave_mode

```
// Enum type for slave mode
// AXI_TRANSACTION_SLAVE - Works at burst level (write data is received at
//                         burst and read data/response is sent in burst)
// AXI_PHASE_SLAVE       - Write data and read data/response is serviced
//                         at phase level
   typedef enum bit
{
      AXI_TRANSACTION_SLAVE = 1'b0,
      AXI_PHASE_SLAVE       = 1'b1
} axi_slave_mode_e;


// Slave mode selection : Default is transaction-level slave
axi_slave_mode_e slave_mode = AXI_TRANSACTION_SLAVE;
```

The default AXI_TRANSACTION_SLAVE mode “saves up” an entire data burst and modifies the slave test program internal memory in zero time for the whole burst. Therefore, a read from internal memory is buffered at the beginning of the read burst for the whole burst. The buffered read data is then transmitted over the protocol signals to the master on a phase-by-phase (beat-by-beat) basis. For a write, the write data burst is buffered on a phase-by-phase (beat-by-beat) basis for the whole burst. Only at the end of the write burst are the buffered contents written to the internal memory.

The AXI_PHASE_SLAVE mode changes the slave test program internal memory on each data phase (beat). Therefore, a read from the internal memory occurs only when the read data phase (beat) actually starts on the protocol signals. For a write, data is written to the internal memory as soon as each individual write data phase (beat) completes.

Note

In addition to the above functions, you can configure other aspects of the AXI3 Slave BFM by using the functions: “*set_config()*” on page 96 and “*get_config()*” on page 98.

Using the AXI3 Basic Slave Test Program API

As described in the AXI3 Basic Slave API Definition section, there are a set of tasks and functions that you can use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing. However, consider the following configurations when using the slave test program.

- *slave_mode* – The read and write channel interaction can cause simultaneous read and write transactions to occur at the same address. With the default *slave_mode* setting the

read transaction data burst is buffered at the start of the burst and the write data burst is buffered at the end of the burst. This can result in the read data being stale at the time it is transmitted over the protocol signals. If this is an undesirable feature, then set the *Slave_mode* to be AXI_PHASE_SLAVE.

- *slave_ready_delay_mode* – By default the handshake *READY signal will always follow, or be simultaneous with, the *VALID signal. By configuring the *delay_mode* to be AXI_TRANS2READY *READY before *VALID scenarios can be achieved.

Advanced AXI3 Slave API Definition

Note

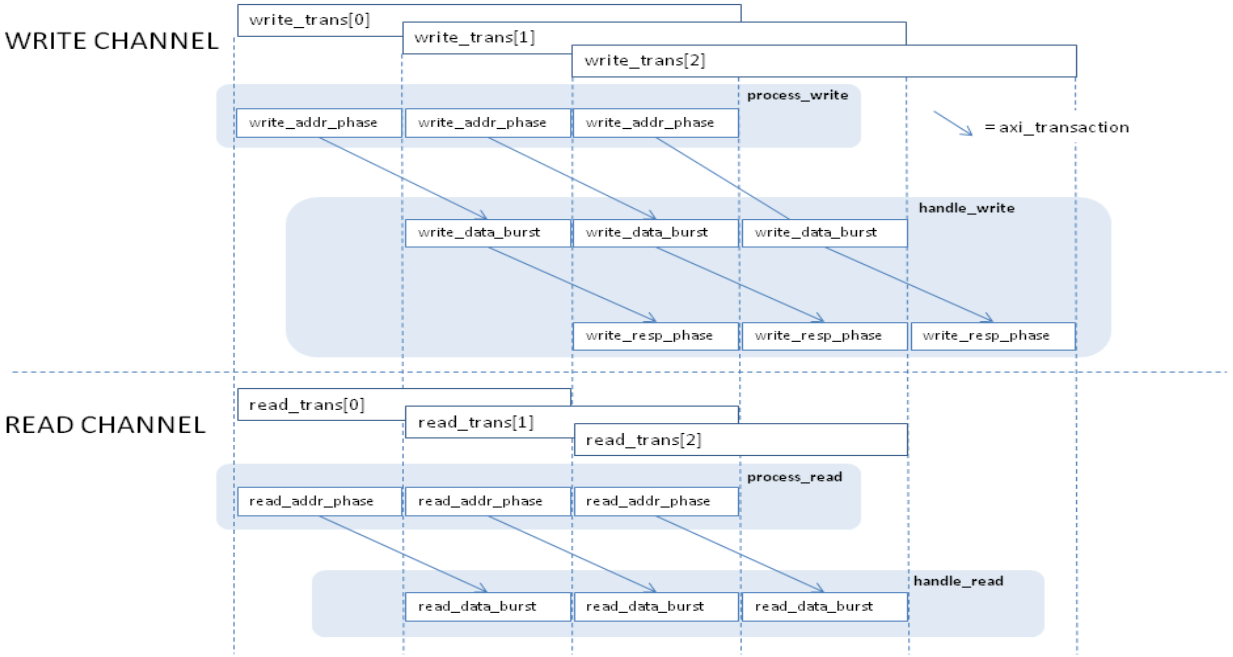
You are not required to edit the following Advance Slave API unless you require a different response than the default (OKAY) response.

The remaining section of this tutorial presents a walk-through of the Advanced Slave API in the slave test program. It consists of four main tasks, *process_read*, *process_write*, *handle_read* and *handle_write* in the slave test program, as shown in Figure 6-5.

The Advanced Slave API is capable of handling pipelined transactions. Pipelining can occur when a transaction starts before a previous transaction has completed. Therefore, a write transaction that starts before a previous write transaction has completed can be pipelined. Figure 6-5 shows the write channel having three concurrent *write_trans* transactions, whereby the *write_addr_phase[2]*, *write_data_burst[1]*, and *write_response_phase[0]* are concurrently active on the write address, data, and response channels, respectively.

Similarly, a read transaction that starts before a previous read transaction has completed can be pipelined. Figure 6-5 shows the read channel having two concurrent *read_trans* transactions, whereby the *read_addr_phase[1]* and *read_data_burst[0]* are concurrently active on the read address and data channels, respectively.

Figure 6-5. Slave Test Program Advanced API Tasks



In an initial block, the slave test program configures the maximum number of outstanding read and write transactions. The slave test program starts the processing of any read or write transactions simultaneously in a fork-join block, as demonstrated in Example 6-26.

Example 6-26. Initialization and Transaction Processing

```
initial
begin

  // Traffic generation
  fork
    process_read;
    process_write;
  join
end
```

The *process_read* task loops forever, processing read transactions as they occur from the master. It defines a local transaction variable *read_trans* of type *axi_transaction* to store a record of the read transaction while it is being processed. It then uses the Slave BFM function *create_slave_transaction()* to create a read transaction and assign it to the local *read_trans* record.

The *set_read_address_ready_delay()* function is called to configure the delay for the ARREADY signal before getting the read address phase using the Slave BFM *get_read_addr_phase()* task.

The subsequent *fork-join_none* block performs a nonblocking statement so that the *process_read* task can begin again to create another read transaction and get another read address phase before the current read transaction has completed. This permits concurrent read transactions to occur if the master issues a series of read address phases before any previous read transactions have completed.

In the *fork-join_none* block, the *read_trans* record is passed into the *handle_read()* function with the variable *t*.

Example 6-27. process_read

```
// Task : process_read
// This method keep receiving read address phase and calls another
// method to process received transaction.
task process_read;
  forever
  begin
    axi_transaction read_trans;

    read_trans = bfm.create_slave_transaction();
    set_read_address_ready_delay(read_trans);
    bfm.get_read_addr_phase(read_trans);

    fork
      begin
        automatic axi_transaction t = read_trans;
        handle_read(t);
      end
    join_none
    #0;
  end
endtask
```

The *set_read_data_ready()* function calls the *set_data_valid_delay()* function in the slave BFM. It configures the delay for the assertion of ARVALID signal for each read data phase (beat) of a read burst.

Example 6-28. set_read_data_ready()

```
// Function : set_read_data_valid_delay
// This is used to set read response phase valid delays to start
// driving read data/response phases after specified delay.
function void set_read_data_valid_delay(axi_transaction trans);
  for (int i = 0; i < trans.data_valid_delay.size(); i++)
    trans.set_data_valid_delay(i, i);
endfunction
```

The *handle_read* task gets the data from the internal memory in either bursts or phases depending on the *slave_mode* configuration. Its *read_trans* argument contains the record of the read transaction up to the point of this task call, namely the content of the read address phase.

The call to *set_read_data_valid_delay()* configures the assertion of the ARVALID signal delay.

The task then loops for the number of addresses defined by calling the *get_read_addr()* helper function in the slave BFM, assigning the local *mem_data* variable with read data by calling the *do_byte_read()* function.

The slave BFM helper function *set_read_data()* then fills a complete read data bus width of data into the slave BFM *data_words[]* array. At this point, a read data phase is executed on to the read data channel if the *slave_mode* setting is AXI_PHASE_SLAVE; otherwise, it executes a complete read data burst only when the slave BFM *data_words[]* array contains a complete burst of read data.

Example 6-29. handle_read

```
// Task : handle_read
// This method reads data from memory and send read data/response
// either at burst or phase level depending upon slave working
// mode.
task automatic handle_read(input axi_transaction read_trans);
  addr_t addr[];
  bit [7:0] mem_data[];

  set_read_data_valid_delay(read_trans);

  for(int i = 0; bfm.get_read_addr(read_trans, i, addr); i++)
  begin
    mem_data = new[addr.size()];
    for (int j = 0; j < addr.size(); j++)
      mem_data[j] = do_byte_read(addr[j]);

    bfm.set_read_data(read_trans, i, addr, mem_data);

    if (slave_mode == AXI_PHASE_SLAVE)
      bfm.execute_read_data_phase(read_trans, i);
  end
    if (slave_mode == AXI_TRANSACTION_SLAVE)
      bfm.execute_read_data_burst(read_trans);
endtask
```

The processing of write transactions in the slave test program works in a similar way as that described for read transactions. Processing a write transaction requires both a *process_write* task and a *handle_write* task.

The main difference is that the write transaction handling gets the write data burst and stores it in the slave test program *internal memory* depending on the *slave_mode* setting, adhering to the state of the write strobes signal WSTRB. There is also an additional write response phase that is required for the AXI3 write channel.

Example 6-30. process_write

```
// Task : process_write
// This method keep receiving write address phase and calls another
// method to process received transaction.
task process_write;
  forever
  begin
    axi_transaction write_trans;

    write_trans = bfm.create_slave_transaction();
    set_write_address_ready_delay(write_trans);
    bfm.get_write_addr_phase(write_trans);

    fork
      begin
        automatic axi_transaction t = write_trans;
        handle_write(t);
      end
    join_none
    #0;
  end
endtask
```

Example 6-31. handle_write

```
// Task : handle_write
// This method receive write data burst or phases for write
// transaction depending upon slave working mode, write data to
// memory and then send response
task automatic handle_write(input axi_transaction write_trans);
  addr_t addr[];
  bit [7:0] data[];
  bit last;

  set_write_data_ready_delay(write_trans);

  if (slave_mode == AXI_TRANSACTION_SLAVE)
        begin
    bfm.get_write_data_burst(write_trans);

    for( int i = 0; bfm.get_write_addr_data(write_trans,
         i, addr, data); i++ )
    begin
      for (int j = 0; j < addr.size(); j++)
         do_byte_write(addr[j], data[j]);
    end
  end
  else
  begin
    for(int i = 0; (last == 1'b0); i++)
    begin
      bfm.get_write_data_phase(write_trans, i, last);

      void'(bfm.get_write_addr_data(write_trans, i, addr, data));
      for (int j = 0; j < addr.size(); j++)
        do_byte_write(addr[j], data[j]);
    end
  end

  set_wr_resp_valid_delay(write_trans);
  bfm.execute_write_response_phase(write_trans);
endtask
```

AXI4 BFM Slave Test Program

The slave test program is a memory model that contains two APIs: a Mentor VIP – Intel FPGA Edition AXI4 Basic Slave API Definition and an AXI4 Advanced Slave API Definition.

The Mentor VIP – Intel FPGA Edition AXI4 Basic Slave API Definition allows you to create a wide range of stimulus scenarios to test a master DUT. This API definition simplifies the creation of slave stimulus based on the default response of OKAY to master read and write transactions. The AXI4 Advanced Slave API Definition allows you to create additional response scenarios to transactions. For example, a successful exclusive transaction requires an EXOKAY response.

Note

For a complete code example of this slave test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi4\qsys-examples\ex1_back_to_back_sv\slave_test_program.sv

Mentor VIP – Intel FPGA Edition AXI4 Basic Slave API Definition

The Basic Slave Test Program API contains the following elements:

- Functions that read and write a byte of data to Internal Memory include *do_byte_read()* and *do_byte_write()*, respectively.
- Functions *set_read_data_valid_delay()* and *set_wr_resp_valid_delay()* to configure the delay of the read data channel RVALID, and write response channel BVALID signals, respectively.
- Variables *m_rd_addr_phase_ready_delay* and *m_wr_addr_phase_ready_delay* to configure the delay of the read/write address channel ARVALID/AWVALID signals, and *m_wr_data_phase_ready_delay* to configure the delay of the write response channel BVALID signal.
- A *slave_mode* variable to configure the behavior of reading and writing to the internal memory.
- A *slave_ready_delay_mode* variable to configure the behavior of the handshake signals *VALID to *READY delay.

Internal Memory

The internal memory for the slave is defined as a sparse array of 8 bits, so that each byte of data is stored as an address/data pair.

Example 6-32. Internal Memory

```
// Storage for a memory
bit [7:0] mem [*];
```

do_byte_read()

The *do_byte_read()* function, when called, will read a data byte from the Internal Memory *mem*, given an address location as shown below.

You can edit this function to modify the way the read data is extracted from the internal memory.

Example 6-33. do_byte_read()

```
// Function : do_byte_read
// Function to provide read data byte from memory at
// particular input address
function bit[7:0] do_byte_read(addr_t addr);
   return mem[addr];
endfunction
```

do_byte_write()

The *do_byte_write()* function, when called, writes a data byte to the Internal Memory *mem*, given an address location as shown below.

You can edit this function to modify the way the write data is stored in the internal memory.

Example 6-34. do_byte_write()

```
// Function : do_byte_write
// Function to write data byte to memory at particular
// input address
function void do_byte_write(addr_t addr, bit [7:0] data);
   mem[addr] = data;
endfunction
```

m_rd_addr_phase_ready_delay

The *m_rd_addr_phase_ready_delay* variable holds the ARREADY signal delay. The delay value extends the length of the read address phase by a number of ACLK cycles. The starting point of the delay is determined by the *slave_ready_delay_mode* variable configuration.

Example 6-35 shows the ARREADY signal delayed by two ACLK cycles. You can edit this variable to change the ARREADY signal delay.

Example 6-35. m_rd_addr_phase_ready_delay

```
// Variable : m_rd_addr_phase_ready_delay
int m_rd_addr_phase_ready_delay = 2;
```

m_wr_addr_phase_ready_delay

The *m_wr_addr_phase_ready_delay* variable holds the AWREADY signal delay. The delay value extends the length of the write address phase by a number of ACLK cycles. The starting point of the delay is determined by the *slave_ready_delay_mode* variable configuration.

Example 6-36 shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.

Example 6-36. m_wr_addr_phase_ready_delay

```
// Variable : m_wr_addr_phase_ready_delay
int m_wr_addr_phase_ready_delay = 2;
```

m_wr_data_phase_ready_delay

The *m_wr_data_phase_ready_delay* variable holds the WREADY signal delay. The delay value extends the length of each write data phase (beat) in a write data burst by a number of ACLK cycles. The starting point of the delay is determined by the *slave_ready_delay_mode* variable configuration.

Example 6-37 shows the WREADY signal delayed by two ACLK cycles. You can edit this function to change the WREADY signal delay.

Example 6-37. m_wr_data_phase_ready_delay

```
// Variable : m_wr_data_phase_ready_delay
int m_wr_data_phase_ready_delay = 2;
```

set_read_data_valid_delay()

The *set_read_data_valid_delay()* function, when called, configures the RVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of each read data phase (beat) in a read data burst. The delay value of the RVALID signal, for each read data phase, is stored in an array element of the *data_valid_delay* transaction field.

Example 6-38 shows the RVALID signal delay incrementing by an ACLK cycle between each read data phase for the length of the burst. You can edit this function to change the RVALID signal delay.

Example 6-38. set_read_data_valid_delay()

```
// Function : set_read_data_valid_delay
// This is used to set read response phase valid delays to start
// driving read data/response phases after specified delay.
function void set_read_data_valid_delay(axi4_transaction trans);
  for (int i = 0; i < trans.data_valid_delay.size(); i++)
    trans.set_data_valid_delay(i, i);
endfunction
```

set_wr_resp_valid_delay()

The *set_wr_resp_valid_delay()* function, when called, configures the BVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of the write response phase. The delay value of the BVALID signal is stored in the *write_response_valid_delay* transaction field.

Example 6-39 shows the BVALID signal delay set to two ACLK cycles. You can edit this function to change the BVALID signal delay.

Example 6-39. set_wr_resp_valid_delay()

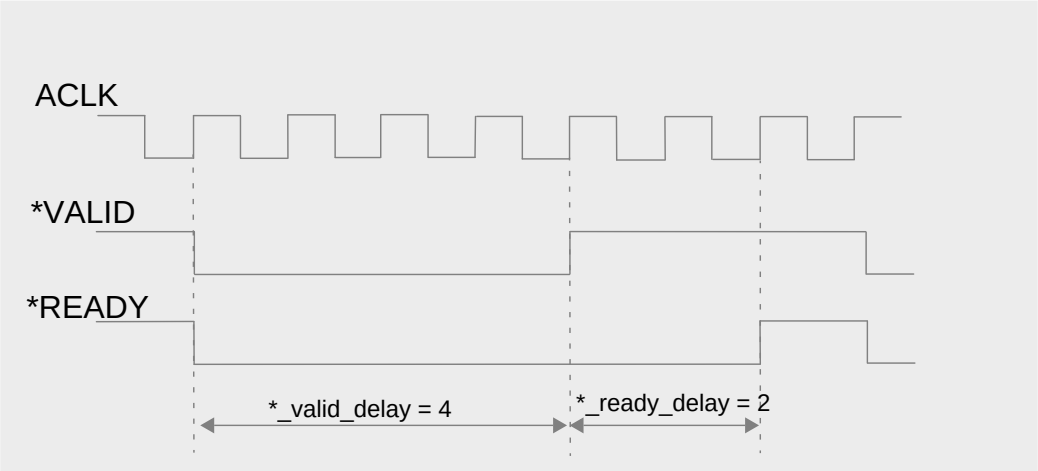
```
// Function : set_wr_resp_valid_delay
// This is used to set write response phase valid delay to start
// driving write response phase after specified delay.
function void set_wr_resp_valid_delay(axi4_transaction trans);
   trans.set_write_response_valid_delay(2);
endfunction
```

slave_ready_delay_mode

The *slave_ready_delay_mode* variable holds the configuration that defines the starting point of any delay applied to the *READY signals. You can configure it to the enumerated type values of AXI4_VALID2READY (default) or AXI4_TRANS2READY.

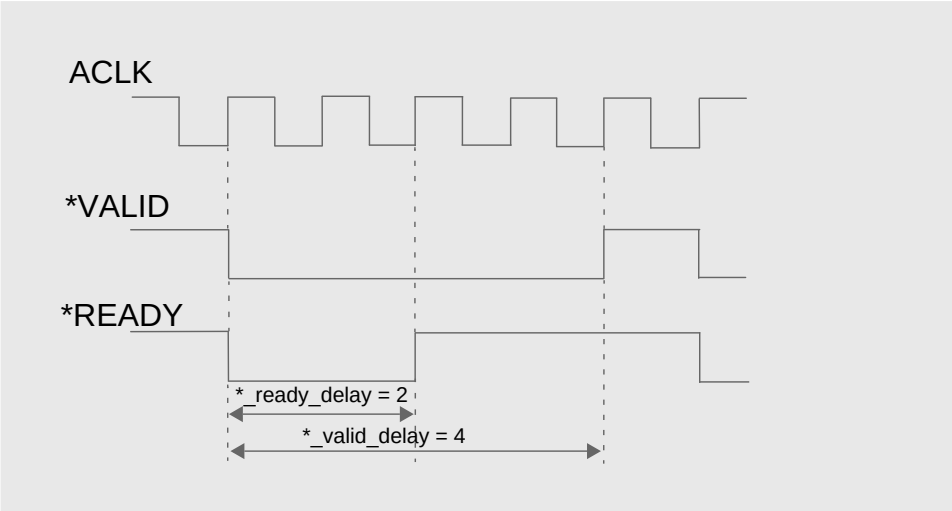
The default configuration (*slave_ready_delay_mode = AXI4_VALID2READY*) corresponds to the delay measured from the positive edge of ACLK when *VALID is asserted. Figure 6-6 shows how to achieve a *VALID before *READY handshake.

Figure 6-6. slave_ready_delay_mode = AXI4_VALID2READY



The nondefault configuration (*slave_ready_delay_mode = AXI4_TRANS2READY*) corresponds to the delay measured from the completion of a previous transaction phase (*VALID and *READY both asserted). Figure 6-7 shows how to achieve a *READY before *VALID handshake.

Figure 6-7. slave_ready_delay_mode = AXI4_TRANS2READY



Example 6-40 shows the configuration of the *slave_ready_delay_mode* to its default value.

Example 6-40. slave_ready_delay_mode

```
// Enum type for slave ready delay mode
  // AXI4_VALID2READY - Ready delay for a phase will be applied from
  //                    start of phase (Means from when VALID is asserted).
  // AXI4_TRANS2READY - Ready delay will be applied from the end of
  //                     previous phase. This might result in ready before
valid.
  typedef enum bit
  {
    AXI4_VALID2READY = 1'b0,
    AXI4_TRANS2READY = 1'b1
  } axi4_slave_ready_delay_mode_e;

// Slave ready delay mode seclection : default it is AXI4_VALID2READY
axi4_slave_ready_delay_mode_e slave_ready_delay_mode = AXI4_VALID2READY;
```

slave_mode

There is a *slave_mode* transaction field that you configure to control the behavior of reading and writing to Internal Memory. It has two modes: AXI4_TRANSACTION_SLAVE and AXI4_PHASE_SLAVE.

Example 6-41. slave_mode

```
// Enum type for slave mode
// AXI4_TRANSACTION_SLAVE - Works at burst level (write data is received
// at burst and read data/response is sent in burst)
// AXI4_PHASE_SLAVE       - Write data and read data/response is serviced
//                          at phase level
   typedef enum bit
{
      AXI4_TRANSACTION_SLAVE = 1'b0,
      AXI4_PHASE_SLAVE       = 1'b1
} axi4_slave_mode_e;
// Slave mode selection : Default is transaction-level slave
axi4_slave_mode_e slave_mode = AXI4_TRANSACTION_SLAVE;
```

The default AXI4_TRANSACTION_SLAVE mode “saves up” an entire data burst and modifies the slave test program Internal Memory in zero time for the whole burst. Therefore, a burst read from internal memory is buffered from the beginning of the burst to the end of the burst. The buffered read burst data is then transmitted over the protocol signals to the master on a phase-by-phase (beat-by-beat) basis. For a write, the data burst received over the protocol signals is buffered from the beginning of the burst to the end of the burst. At the end of the write burst, the buffered contents are written to the internal memory.

The AXI4_PHASE_SLAVE mode updates the slave test program Internal Memory on each data phase (beat). Therefore, a read from the internal memory occurs only when the read data phase (beat) actually starts to be transmitted on the protocol signals. For a write, data is written to the internal memory as soon as each write data phase (beat) is received on the protocol signals.

Note

In addition to the previous variables and procedures, you can configure other aspects of the AXI4 Slave BFM by using the procedures *“set_config()”* on page 96 and *“get_config()”* on page 98.

Using the AXI4 Basic Slave Test Program API

There are a set of tasks and functions that you can use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing, as described in the Mentor VIP – Intel FPGA Edition AXI4 Basic Slave API Definition section.

Consider the following configurations when using the slave test program.

- *slave_mode* – The read and write channel interaction can cause simultaneous read and write transactions to occur at the same address. With the default *slave_mode* setting the read transaction data burst is buffered at the start of the burst and the write data burst is buffered at the end of the burst. This can result in the read data being stale at the time it is transmitted over the protocol signals. If this is an undesirable feature, then set the *slave_mode* to AXI4_PHASE_SLAVE.

- *slave_ready_delay_mode* – By default, each channel handshake *READY signal will always follow, or be simultaneous with, the channel *VALID signal. By configuring the *slave_ready_delay_mode* to AXI4_TRANS2READY, *READY before *VALID scenarios can be achieved.

AXI4 Advanced Slave API Definition

Note

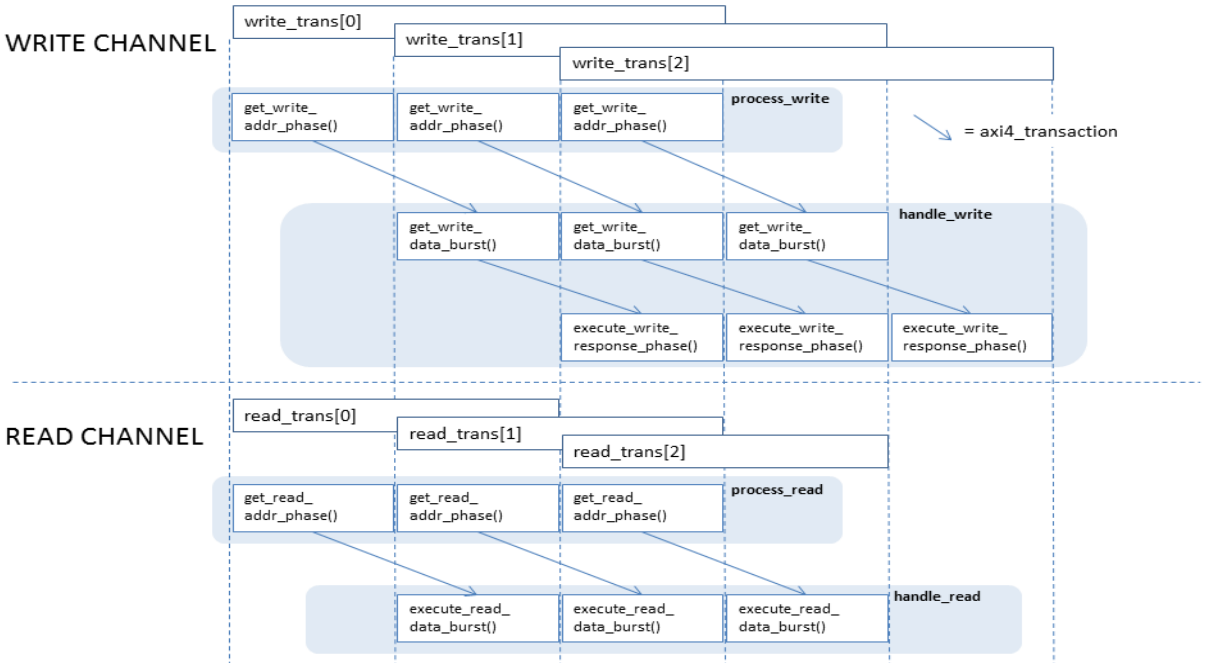
You are not required to edit the following Advance Slave API unless you require a different response than the default (OKAY) response.

The remaining section of this tutorial presents a walk-through of the Advanced Slave API in the slave test program. It consists of four main tasks: *process_read()*, *process_write()*, *handle_read()*, and *handle_write()* in the slave test program, as shown in Figure 6-8. There are additional *handle_write_addr_ready()*, *handle_read_addr_ready()* and *handle_write_data_ready()* tasks to handle the handshake AWREADY, ARREADY and WREADY signals, respectively.

The Advanced Slave API is capable of handling pipelined transactions. Pipelining can occur when a transaction starts before a previous transaction has completed. Therefore, a write transaction that starts before a previous write transaction has completed can be pipelined. Figure 6-8 shows the write channel with three concurrent *write_trans* transactions, whereby the *get_write_addr_phase[2]*, *get_write_data_burst[1]*, and *execute_write_response_phase[0]* are concurrently active on the write address, data, and response channels, respectively.

Similarly, a read transaction that starts before a previous read transaction has completed can be pipelined. Figure 6-8 shows the read channel with two concurrent *read_trans* transactions, whereby the *get_read_addr_phase[1]* and *execute_read_data_burst[0]* are concurrently active on the read address and data channels, respectively.

Figure 6-8. Slave Test Program Advanced API Tasks



Initial Block

In an *initial* block, the slave test program configures the maximum number of outstanding read and write transactions. The slave test program then starts the processing of any read or write transactions, and the handling of the channel *READY signals in a fork-join block, as shown in Example 6-42.

Example 6-42. Initialization and Transaction Processing

```
initial
begin

   // Traffic generation
   fork
      process_read;
      process_write;
      handle_write_addr_ready;
      handle_read_addr_ready;
      handle_write_data_ready;
   join
end
```

process_read()

The *process_read()* task loops forever, processing read transactions as they occur from the master. A local transaction variable *read_trans* of type *axi4_transaction* is defined to hold a record of the read transaction while it is being processed. A slave transaction is created by calling the *create_slave_transaction()* function and assigned to the *read_trans* record.

The subsequent *fork-join_none* block performs a nonblocking statement so that the *process_read()* task can begin again to create another read transaction record and get another read address phase before the current read transaction has completed. This permits concurrent read transactions to occur if the master issues a series of read address phases before any previous read transactions have completed.

In the *fork-join_none* block, the *read_trans* record is passed into the *handle_read()* function via the variable *t*.

Example 6-43. process_read()

```
// Task : process_read
// This method keep receiving read address phase and calls another
// method to process received transaction.
task process_read;
  forever
  begin
    axi4_transaction read_trans;

    read_trans = bfm.create_slave_transaction();
    bfm.get_read_addr_phase(read_trans);

    fork
      begin
        automatic axi4_transaction t = read_trans;
        handle_read(t);
      end
    join_none
    #0;
  end
endtask
```

handle_read()

The *handle_read()* task gets the data from the Internal Memory as a burst or a phase (beat), depending on the *slave_mode* configuration. The *read_trans* argument contains the record of the read transaction up to the point of this task call, namely the content of the read address phase.

The call to *set_read_data_valid_delay()* configures the RVALID signal delay for each phase (beat).

In a *loop*, the call to the *get_read_addr()* helper function returns the actual address *addr* for a particular byte location. This byte address is used to read the data byte from Internal Memory with the call to the *do_byte_read()* function, assigning the local *mem_data* variable with read data *do_byte_read()*. The call to the *set_read_data()* helper function sets the byte within the read transaction record. The loop continues reading and setting the read data from internal memory for the whole of the read data phase (beat).

If the *slave_mode* configuration is set to the default of AXI4_TRANSACTION_SLAVE, then the loop continues until the read data has been set for the whole burst. Otherwise, the individual read data phase is executed over the protocol signals by calling the *execute_read_data_phase()*. After the for loop is complete, *execute_read_data_burst()* is called for the default configuration of *slave_mode*, and the read burst is executed over the protocol signals.

Example 6-44. handle_read

```
// Task : handle_read
// This method reads data from memory and send read data/response
// either at burst or phase level depending upon slave working
// mode.
task automatic handle_read(input axi4_transaction read_trans);
   addr_t addr[];
   bit [7:0] mem_data[];

   set_read_data_valid_delay(read_trans);

   for(int i = 0; bfm.get_read_addr(read_trans, i, addr); i++)
   begin
      mem_data = new[addr.size()];
      for (int j = 0; j < addr.size(); j++)
         mem_data[j] = do_byte_read(addr[j]);

      bfm.set_read_data(read_trans, i, addr, mem_data);

      if (slave_mode == AXI4_PHASE_SLAVE)
         bfm.execute_read_data_phase(read_trans, i);
   end
   if (slave_mode == AXI4_TRANSACTION_SLAVE)
      bfm.execute_read_data_burst(read_trans);
endtask
```

process_write()

The processing of write transactions in the slave test program works in a similar way as that previously described for the *process_read()* task.

Example 6-45. process_write

```
// Task : process_write
// This method keep receiving write address phase and calls another
// method to process received transaction.
task process_write;
  forever
  begin
    axi4_transaction write_trans;

    write_trans = bfm.create_slave_transaction();
    bfm.get_write_addr_phase(write_trans);

    fork
      begin
        automatic axi4_transaction t = write_trans;
        handle_write(t);
      end
    join_none
    #0;
  end
endtask
```

handle_write()

The *handle_write()* task works in a similar way as that previously described for the *handle_read()* task. The main difference is that the write transaction handling gets the write data burst and stores it in the slave test program Internal Memory depending on the *slave_mode* setting, and adhering to the state of the WSTRB write strobes signal. There is an additional write response phase that is required for the write response channel, as shown in Example 6-46.

Example 6-46. handle_write()

```
// Task : handle_write
// This method receive write data burst or phases for write
// transaction depending upon slave working mode, write data to
// memory and then send response
task automatic handle_write(input axi4_transaction write_trans);
   addr_t addr[];
   bit [7:0] data[];
   bit last;

   if (slave_mode == AXI4_TRANSACTION_SLAVE)
   begin
      bfm.get_write_data_burst(write_trans);

      for( int i = 0; bfm.get_write_addr_data(write_trans,
            i, addr, data); i++ )
      begin
         for (int j = 0; j < addr.size(); j++)
            do_byte_write(addr[j], data[j]);
      end
   end
   else
   begin
      for(int i = 0; (last == 1'b0); i++)
      begin
         bfm.get_write_data_phase(write_trans, i, last);
         void'(bfm.get_write_addr_data(write_trans, i, addr, data));
         for (int j = 0; j < addr.size(); j++)
            do_byte_write(addr[j], data[j]);
      end
   end
   set_wr_resp_valid_delay(write_trans);
   bfm.execute_write_response_phase(write_trans);
endtask
```

handle_write_addr_ready()

The *handle_write_addr_ready()* task handles the AWREADY signal for the write address channel. In a forever loop, it delays the assertion of the AWREADY signal based on the settings of the *slave_ready_delay_mode* and *m_wr_resp_phase_ready_delay* as shown in Example 6-47.

If the *slave_delay_ready_mode* = AXI4_VALID2READY, then the AWREADY signal is deasserted using the nonblocking call to the *execute_write_data_ready()* task and waits for a

write channel address phase to occur with a call to the blocking *get_write_addr_cycle()* task. A received write address phase indicates that the AWVALID signal has been asserted, triggering the starting point for the delay of the AWREADY signal by the number of ACLK cycles defined by *m_wr_addr_phase_ready_delay*. Another call to the *execute_write_addr_ready()* task to assert the AWREADY signal completes the AWREADY handling. The *seen_valid_ready* flag is set to indicate the end of a address phase when both AWVALID and AWREADY are asserted.

If the *slave_delay_ready_mode* = AXI4_TRANS2READY, then a check of the *seen_valid_ready* flag is performed to indicate that a previous write address phase has completed. If a write address phase is still active (indicated by either AWVALID or AWREADY not asserted), then the code waits until the previous write address phase has completed. The AWREADY signal is then deasserted using the nonblocking call to the *execute_write_addr_ready()* task and waits for the number of ACLK cycles defined by *m_wr_addr_phase_ready_delay*. A nonblocking call to the *execute_write_addr_ready()* task to assert the AWREADY signal completes the AWREADY handling. The *seen_valid_ready* flag is cleared to indicate that only AWREADY has been asserted.

Example 6-47. handle_write_addr_ready()

```
// Task : handle_write_addr_ready
// This method assert/de-assert the write address channel ready signal.
// Assertion and de-assertion is done based on m_wr_addr_phase_ready_delay
task automatic handle_write_addr_ready;
   bit seen_valid_ready;

   int tmp_ready_delay;
   int tmp_config_num_outstanding_wr_phase;

   axi4_slave_ready_delay_mode_e tmp_mode;

   forever
   begin
      tmp_config_num_outstanding_wr_phase = bfm.get_config
         (AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE);
      while
      (
         (tmp_config_num_outstanding_wr_phase >=
          bfm.get_config(AXI4_CONFIG_MAX_OUTSTANDING_WR) &&
         (bfm.get_config(AXI4_CONFIG_MAX_OUTSTANDING_WR > 0)
      )
      begin
         bfm.wait_on(AXI4_CLOCK_POSEDGE);
         tmp_config_num_outstanding_wr_phase = bfm.get_config
            (AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE);
      end
      wait(m_wr_addr_phase_ready_delay > 0);
      tmp_ready_delay = m_wr_addr_phase_ready_delay;
      tmp_mode        = slave_ready_delay_mode;

      if (tmp_mode == AXI4_VALID2READY)
      begin
         fork
```

```
            bfm.execute_write_addr_ready(1'b0);
         join_none

         bfm.get_write_addr_cycle;
         repeat(tmp_ready_delay - 1) bfm.wait_on(AXI4_CLOCK_POSEDGE);

         bfm.execute_write_addr_ready(1'b1);
         seen_valid_ready = 1'b1;
      end
      else  // AXI4_TRANS2READY
      begin
         if (seen_valid_ready == 1'b0)
         begin
            do
               bfm.wait_on(AXI4_CLOCK_POSEDGE);
            while (!((bfm.AWVALID === 1'b1) && (bfm.AWREADY === 1'b1)));
         end

         fork
            bfm.execute_write_addr_ready(1'b0);
         join_none

         repeat(tmp_ready_delay) bfm.wait_on(AXI4_CLOCK_POSEDGE);

         fork
            bfm.execute_write_addr_ready(1'b1);
         join_none
         seen_valid_ready = 1'b0;
      end
   end
endtask
```

handle_read_addr_ready()

The *handle_read_addr_ready()* task handles the ARREADY signal for the read address channel. In a forever loop, it delays the assertion of the ARREADY signal based on the settings of the *slave_ready_delay_mode* and *m_rd_addr_phase_ready_delay*. The *handle_read_addr_ready()* task code is similar in operation to the *handle_write_addr_ready()* task.

handle_write_data_ready()

The *handle_write_data_ready()* task handles the WREADY signal for the write data channel. In a forever loop, it delays the assertion of the WREADY signal based on the settings of the *slave_ready_delay_mode* and *m_wr_data_phase_ready_delay*. The *handle_write_data_ready()* task code is similar in operation to the *handle_write_addr_ready()* task.

Chapter 7
VHDL API Overview

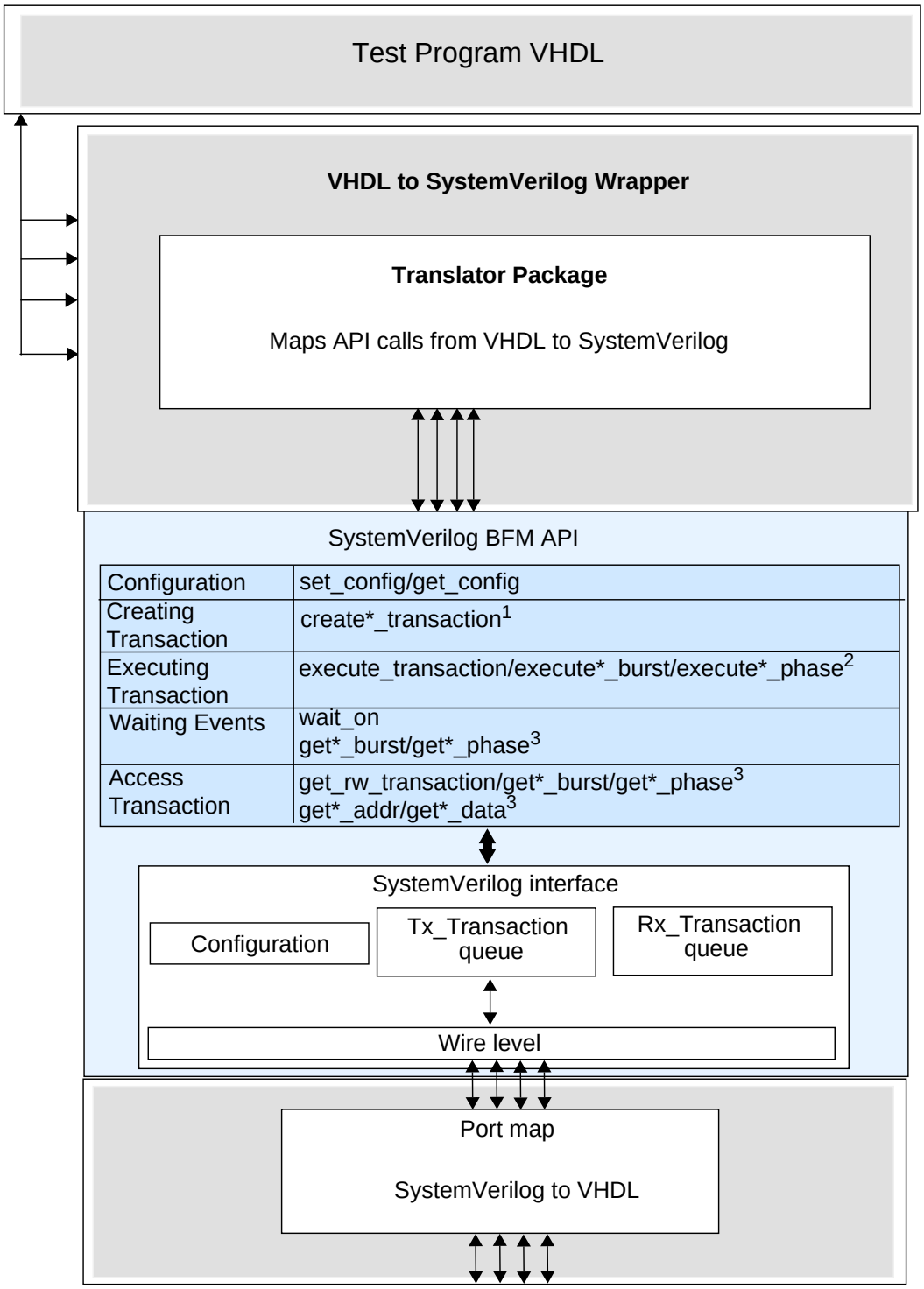
This chapter describes the VHDL Application Programming Interface (API) procedures for all the BFM (master, slave, and monitor) components. For each BFM, you can configure protocol transaction fields that execute on the protocol signals and control the operational transaction fields that permit delays between the handshake signals for each of the five address, data, and response channels.

In addition, each BFM API has procedures that wait for certain events to occur on the system clock and reset signals, and procedures to get and set information about a particular transaction.

Note

The VHDL API is built on the SystemVerilog API. An internal VHDL to SystemVerilog (SV) wrapper casts the VHDL BFM API procedure calls to the SystemVerilog BFM API tasks and functions.

Figure 7-1. VHDL BFM Internal Structure



Notes: 1. Refer to the create*_transaction()
2. Refer to the execute_transaction(), execute*_burst(), execute*_phase()
3. Refer to the get*()

Configuration

Configuration sets timeout delays, error reporting, and other attributes of the BFM.

Each BFM has a *set_config()* procedure that sets the configuration of the BFM. Refer to the individual BFM API for valid details. Each BFM has a *get_config()* procedure that returns the configuration of the BFM. Refer to the individual BFM API for details.

set_config()

For example, the following test program code sets the burst timeout factor for a transaction in the master BFM:

```
-- Setting the burst timeout factor to 1000
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi_tr_if_0(bfm_index))
```

In the above example, the *bfm_index* specifies the BFM.

Note

The above test program code segment is for AXI3 BFMs. Substitute the AXI_CONFIG_BURST_TIMEOUT_FACTOR enumeration with AXI4_CONFIG_BURST_TIMEOUT_FACTOR, and the *axi_tr_if_0* path name with *axi4_tr_if_0* for AXI4 BFMs.

get_config()

For example, the following test program code gets the protocol signal hold time in the master BFM:

```
-- Getting the burst timeout factor
get_config(AXI_CONFIG_HOLD_TIME, config_value, bfm_index,
      axi_tr_if_0(bfm_index))
```

In the above example, the *bfm_index* specifies the BFM.

Note

The above test program code segment is for AXI3 BFMs. Substitute the AXI_CONFIG_HOLD_TIME enumeration with AXI4_CONFIG_HOLD_TIME, and the *axi_tr_if_0* path name with *axi4_tr_if_0* for AXI4 BFMs.

Creating Transactions

To transfer information between a master BFM and slave DUT over the protocol signals, you must create a transaction in the master test program. Similarly, to transfer information between

a master DUT and a slave BFM, you must create a transaction in the slave test program. To monitor the transfer of information using a monitor BFM, you must create a transaction in the monitor test program.

Creating a transaction also creates a Transaction Record that exists for the life of the transaction. This transaction record can be accessed by the BFM test program during the life of the transaction as it transfers information between the master and slave.

Transaction Record

The transaction record contains transaction fields. There are two main types of transaction fields, *protocol* and *operational*.

Protocol fields hold transaction information that is transferred over the protocol signals. For example, the *prot* field is transferred over the AWPROT protocol signals during a write transaction.

Operational fields hold information about how and when the transaction is transferred. Their content is not transferred over protocol signals. For example, the *operation_mode* field controls the blocking/nonblocking operation of the transaction, but is not transferred over the protocol signals.

AXI3 Transaction Definition

The transaction record exists as a SystemVerilog class definition in each BFM. Example 7-1 shows the definition of the *axi_transaction* class members that form the transaction record.

Example 7-1. AXI3 Transaction Definition

```
// Global Transaction Class
class axi_transaction;
    // Protocol
    bit [((`MAX_AXI_ADDRESS_WIDTH) - 1):0]  addr;
    axi_size_e size;
    axi_burst_e burst;
    axi_lock_e lock;
    axi_cache_e cache;
    axi_prot_e prot;
    bit [((`MAX_AXI_ID_WIDTH) - 1):0]  id;
    bit [3:0] burst_length;
    bit [(((`MAX_AXI_RDATA_WIDTH > `MAX_AXI_WDATA_WIDTH) ?
`MAX_AXI_RDATA_WIDTH : `MAX_AXI_WDATA_WIDTH)) - 1):0] data_words [];
    bit [(((`MAX_AXI_WDATA_WIDTH / 8)) - 1):0] write_strobes [];
    axi_response_e resp[];
    bit [7:0] addr_user;
    axi_rw_e read_or_write;
    int address_valid_delay;
    int data_valid_delay[];
    int write_response_valid_delay;
    int address_ready_delay;
    int data_ready_delay[];
    int write_response_ready_delay;

    // Housekeeping
    bit gen_write_strobes = 1'b1;
    axi_operation_mode_e  operation_mode  = AXI_TRANSACTION_BLOCKING;
    axi_delay_mode_e      delay_mode      = AXI_VALID2READY;
    axi_write_data_mode_e write_data_mode = AXI_DATA_WITH_ADDRESS;
    bit data_beat_done[];
    bit transaction_done;

...

endclass
```

Note

The *axi_transaction* class code above is shown for information only. Access to each transaction record during its lifetime is performed via the various *set*()* and *get*()* procedures detailed later in this chapter.

AXI4 Transaction Definition

The transaction record exists as a SystemVerilog class definition in each BFM. Example 7-2 shows the definition of the *axi4_transaction* class members that form the transaction record.

Example 7-2. AXI4 Transaction Definition

```
// Global Transaction Class
class axi4_transaction;
    // Protocol
    axi4_rw_e read_or_write;
    bit [((`MAX_AXI4_ADDRESS_WIDTH) - 1):0]  addr;
    axi4_prot_e prot;
    bit [3:0] region;
    axi4_size_e size;
    axi4_burst_e burst;
    axi4_lock_e lock;
    axi4_cache_e cache;
    bit [3:0] qos;
    bit [((`MAX_AXI4_ID_WIDTH) - 1):0] id;
    bit [7:0] burst_length;
    bit [((`MAX_AXI4_USER_WIDTH) - 1):0]  addr_user;
    bit [(((`MAX_AXI4_RDATA_WIDTH > `MAX_AXI4_WDATA_WIDTH) ?
`MAX_AXI4_RDATA_WIDTH : `MAX_AXI4_WDATA_WIDTH)) - 1):0] data_words [];
    bit [(((`MAX_AXI4_WDATA_WIDTH / 8)) - 1):0] write_strobes [];
    axi4_response_e resp[];
    int address_valid_delay;
    int data_valid_delay[];
    int write_response_valid_delay;
    int address_ready_delay;
    int data_ready_delay[];
    int write_response_ready_delay;

    // Housekeeping
    bit gen_write_strobes = 1'b1;
    axi4_operation_mode_e  operation_mode  = AXI4_TRANSACTION_BLOCKING;
    axi4_write_data_mode_e write_data_mode = AXI4_DATA_WITH_ADDRESS;
    bit data_beat_done[];
    bit transaction_done;

...

endclass
```

Note

The *axi4_transaction* class code above is shown for information only. Access to each transaction record during its lifetime is performed via the various *set*()* and *get*()* procedures detailed later in this chapter.

Table 7-1 describes the transaction fields in the transaction record.

Table 7-1. Transaction Fields

Transaction Field	Description
Protocol Transaction Fields	
addr	A bit vector (of length equal to the ARADDR/AWADDR signal bus width) to hold the start *address* of the first transfer (beat) of a transaction. The *addr* value is transferred over the ARADDR or AWADDR signals for a read or write transaction, respectively.
prot	An enumeration to hold the *protection* type of a transaction. The types of *protection* are as follows: **_NORM_SEC_DATA (default) **_PRIV_SEC_DATA **_NORM_NONSEC_DATA **_PRIV_NONSEC_DATA **_NORM_SEC_INST **_PRIV_SEC_INST **_NORM_NONSEC_INST **_PRIV_NONSEC_INST The *prot* value is transferred over the ARPROT or AWPROT signals for a read or write transaction, respectively.
region	(AXI4) A 4-bit vector to hold the *region* identifier of a transaction. The *region* value is transferred over the ARREGION or AWREGION signals for a read or write transaction, respectively.
size	An enumeration to hold the *size* of a transaction. The types of *size* are as follows: **_BYTES_1 **_BYTES_2 **_BYTES_4 **_BYTES_8 **_BYTES_16 **_BYTES_32 **_BYTES_64 **_BYTES_128 The *size* value is transferred over the ARSIZE or AWSIZE signals for a read or write transaction, respectively.

Table 7-1. Transaction Fields (cont.)

Transaction Field	Description
burst	An enumeration to hold the *burst* of a transaction. The types of *burst* are as follows: **_FIXED **_INCR **_WRAP **_BURST_RSVD The *burst* value is transferred over the ARBURST or AWBURST signals for a read or write transaction, respectively.
qos	(AXI4) A 4-bit vector to hold the *Quality of Service* identifier of a transaction. The *qos* value is transferred over the ARQOS or AWQOS signals for a read or write transaction, respectively.
id	A bit vector (of length equal to the ARID/AWID signal bus width) to hold the *identification tag* of a transaction. The *id* value is transferred over the AWID/BID signals for a write transaction and over the ARID/RID signals for a read transaction.
burst_length	A 4-bit (8-bit for AXI4) vector to hold the burst length of a transaction. The *burst_length* value is transferred over the ARLEN or AWLEN signals for a read or write transaction, respectively.
addr_user	A bit vector (of length equal to the ARUSER/AWUSER signal bus width) to hold the address channel *user data* of a transaction. The *addr_data* value is transferred over the ARUSER or AWUSER signals for a read or write transaction, respectively.
data_words	An unsized array of bit vectors (of length equal to the greater of the RDATA/WDATA signal bus widths) to hold the *data words* of the payload. A *data_words* array element is transferred over the RDATA or WDATA signals per beat of the read or write data channel, respectively.
write_strobes	An unsized array of bit vectors (of length equal to the WDATA signal bus width divided by 8) to hold the write strobes. A *write_strobes* array element is transferred over the WSTRB signals per beat of the write data channel.

Table 7-1. Transaction Fields (cont.)

Transaction Field	Description
resp	An unsized enumeration array to hold the *responses* of a transaction. The types of *response* are as follows: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR; A *resp* array element value is transferred over the RRESP signals per beat of the read data channel, and over the BRESP signals for a write transaction, respectively.
Operational Transaction Fields	
read_or_write	An enumeration to hold the *read or write* control flag. The types of *read_or_write* are: **_TRANS_READ **_TRANS_WRITE
address_valid_delay	An integer to hold the delay value of the address channel AWVALID and ARVALID signals (measured in ACLK cycles) for a read or write transaction, respectively.
data_valid_delay	An unsized array of integers to hold the delay values of the data channel WVALID and RVALID signals (measured in ACLK cycles) for a read or write transaction, respectively.
write_response_valid_delay	An integer to hold the delay value of the write response channel BVALID signal (measured in ACLK cycles) for a write transaction.
address_ready_delay	An integer to hold the delay value of the address channel AWREADY and ARREADY signals (measured in ACLK cycles) for a read or write transaction, respectively.
data_ready_delay	An unsized array of integers to hold the delay values of the data channel WREADY and RREADY signals (measured in ACLK cycles) for a read or write transaction, respectively.
write_response_ready_delay	An integer to hold the delay value of the write response channel BREADY signal (measured in ACLK cycles) for a write transaction.
gen_write_strobes	Automatically correct write strobes flag. Refer to Automatic Correction of Byte Lane Strobes for details.

Table 7-1. Transaction Fields (cont.)

Transaction Field	Description
operation_mode	An enumeration to hold the *operation mode* of the transaction. The two types of *operation_mode* are: **_TRANSACTION_NON_BLOCKING **_TRANSACTION_BLOCKING
delay_mode	(AXI3) An enumeration to hold the *delay mode* control flag. The types of *delay_mode* are as follows: AXI_VALID2READY AXI_TRANS2READY Refer to AXI3 BFM Delay Mode for details.
write_data_mode	An enumeration to hold the *write data mode* control flag. The types of *write_data_mode* are as follows: **_DATA_AFTER_ADDRESS The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.
data_beat_done	An unsized bit array to hold the *done* flag for each beat in a read or write data burst when it has completed.
transaction_done	A bit to hold the *done* flag for a transaction when it has completed.

The master BFM API allows you to create a master transaction by providing only the address and burst length arguments for a read, or write, transaction. All other protocol transaction fields automatically default to legal protocol values to create a complete master transaction record.

Refer to the *create_read_transaction()* and *create_write_transaction()* procedures for default protocol read and write transaction field values.

The slave BFM API allows you to create a slave transaction by providing no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the *create_slave_transaction()* procedure for default protocol transaction field values.

The monitor BFM API allows you to create a slave transaction by providing no arguments. All protocol transaction fields automatically default to legal protocol values to create a complete slave transaction record. Refer to the *create_monitor_transaction()* procedure for default protocol transaction field values.

Note

If you change a protocol transaction field value from its default, it is then valid for all future transactions until a new value is set.

create*_transaction()

There are two master BFM API procedures available to create transactions, *create_read_transaction()* and *create_write_transaction()*, a *create_slave_transaction()* slave BFM API procedure, and a *create_monitor_transaction()* monitor BFM API procedure.

For example, to create a simple write transaction with a start address of 1, and a single data phase with a data value of 2, the master BFM test program would contain the following code:

```
-- * = axi| axi4
-- ** = AXI | AXI4
-- Define local variables to hold the transaction ID
-- and data word.
variable tr_id: integer;
variable data_words :  std_logic_vector(**_MAX_BIT_SIZE-1 downto 0);

-- Create a master write transaction and set data_word value
create_write_transaction(1, tr_id, bfm_index, *_tr_if_0(bfm_index));
data_words(31 downto 0) := x"00000200";
set_data_words(data_words, tr_id, bfm_index, *_tr_if_0(bfm_index));
```

For example, to create a simple slave transaction the slave BFM test program would contain the following code:

```
-- Define a local variable write_trans to hold the transaction ID
variable write_trans : integer;

-- Create a slave transaction

create_slave_transaction(write_trans, bfm_index, *_tr_if_0(bfm_index));
```

In the above examples, the *bfm_index* specifies the BFM.

Executing Transactions

Executing a transaction in a master/slave BFM test program initiates the transaction onto the protocol signals. Each master/slave BFM API has execution tasks that push transactions into the BFM internal transaction queues. Figure 7-1 on page 202 illustrates the internal BFM structure.

execute_transaction(), execute*_burst(), execute*_phase()

If the DUT is a slave then the *execute_transaction()* procedure is called in the master BFM test program. If the DUT is a master, then the *execute*_burst()* and *execute*_phase()* procedures are called in the slave BFM test program.

For example, to execute a master write transaction the master BFM test program would contain the following code:

```
-- * = axi| axi4
-- By default the execution of a transaction will block
execute_transaction(tr_id, bfm_index, *_tr_if_2(bfm_index));
```

For example, to execute a slave write response phase, the slave BFM test program would contain the following code:

```
-- * = axi| axi4
-- By default the execution of a phase will block
execute_write_response_phase(write_trans, bfm_index,
*_tr_if_2(bfm_index));
```

In the above example, the *bfm_index* specifies the BFM.

Waiting Events

Each BFM API has procedures that block the test program code execution until an event has occurred.

The *wait_on()* procedure blocks the test program until an ACLK or ARESETn signal event has occurred before proceeding.

The *get*_transaction(), get*_burst(), get*_phase(), get*_cycle()* procedures block the test program code execution until a complete transaction, burst, phase or cycle has occurred, respectively.

wait_on()

For example, a BFM test program can wait for the positive edge of the *ARESETn* signal using the following code:

```
-- * = axi| axi4
-- ** = AXI | AXI4
-- Block test program execution until the positive edge of the clock
wait_on(**_RESET_POSEDGE, bfm_index, *_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the BFM.

get*_transaction(), get*_burst(), get*_phase(), get*_cycle()

For example, a slave BFM test program can use a received write address phase to form the response of the write transaction. The test program gets the write address phase for the transaction by calling the *get_write_addr_phase()* procedure. This task blocks until it has received the address phase, allowing the test program to then call the *execute_write_response_phase()* procedure for the transaction, as shown in the slave BFM test program in Example 7-3 below.

Example 7-3. Slave BFM Test Program Using get_write_addr_phase()

```
-- * = axi| axi4
-- ** = AXI | AXI4
create_slave_transaction(write_trans, bfm_index, *_tr_if_0(bfm_index));
get_write_addr_phase(write_trans, bfm_index, *_tr_if_0(bfm_index));

...

execute_write_response_phase(write_trans, bfm_index, **_PATH_2,
*_tr_if_2(bfm_index));
```

In the above example, the *bfm_index* specifies the BFM.

Note

Not all BFM APIs support the full complement of *get*_transaction(), get*_burst(), get*_phase(), get*_cycle()* tasks. Refer to the individual master, slave, or monitor BFM API for details.

Access Transaction Record

Each BFM API has procedures that can access a complete, or partially complete, Transaction Record. The *set*()* and *get*()* procedures are used in a test program to set and get information from the transaction record.

set*()

For example, to set the WSTRB write strobes signal for the first phase (beat) in the Transaction Record of a write transaction, the master test program would use the *set_write_strobes()* procedure, as shown in the code below.

```
-- * = axi| axi4
set_write_strobes(2, tr_id, bfm_index, *_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the BFM.

get*()

For example, a slave BFM test program uses a received write address phase to get the AWPROT signal value from the Transaction Record, as shown in the following slave BFM test program code.

```
-- * = axi| axi4
-- Wait for a write address phase;
get_write_addr_phase(slave_trans, bfm_index, *axi_tr_if_0(bfm_index));

...

-- Get the AWPROT signal value of the slave transaction
get_prot(prot_value, slave_trans, bfm_index, *axi_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the BFM.

Operational Transaction Fields

Operational transaction fields control the way in which a transaction is executed on the protocol signals. These fields also provide an indicator of when a data phase (beat) or transaction is complete.

Automatic Correction of Byte Lane Strobes

The master BFM permits unaligned and narrow write transfers by using byte lane strobe (WSTRB) signals to indicate which byte lanes contain valid data per data phase (beat).

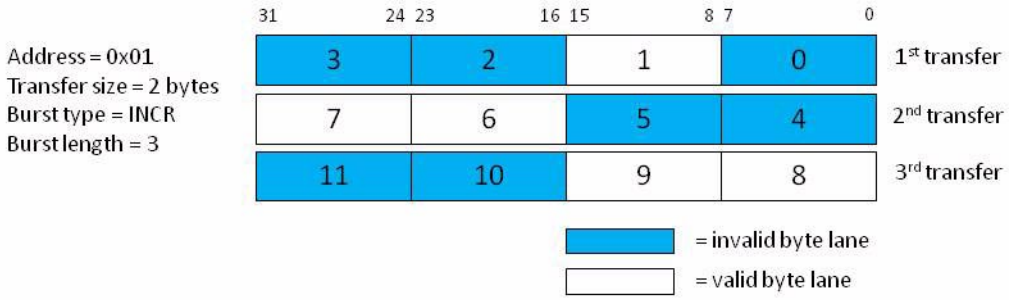
When you create a write transaction in your master BFM test program, the *write_strobes* variable is available to store the write strobe values for each write data phase (beat) in the transaction. To assist you in creating the correct byte lane strobes, automatic correction of any previously set *write_strobes* is performed by default during execution of the write transaction, or write data phase (beat). You can disable this default behavior by setting the transaction field *gen_write_strobes = 0*, which allows any previously set *write_strobes* to pass through uncorrected onto the protocol WSTRB signals. In this mode, with the automatic correction disabled, you are responsible for setting the correct *write_strobes* for the whole transaction.

The automatic correction algorithm performs a bit-wise AND operation on any previously set *write_strobes*. To do the corrections, the automatic correction algorithm uses the equations described in the AMBA AXI Protocol Specification, Version 2.0, Section A3.4.1, that define valid write data byte lanes for legal protocol. Therefore, if you require automatic generation of all *write_strobes*, before the write transaction executes, you must set all *write_strobes* to 1,

indicating that all bytes lanes initially contain valid write data, prior to execution of the write transaction. Automatic correction will then set the relevant *write_strobes* to 0 to produce legal protocol WSTRB signals.

For example, Figure 7-2 demonstrates byte lanes that can contain valid data for a write transaction that has a start address = 0x01, size = 0b001 (2 bytes), type = INCR and length = 0b0010 (3 beats), for a 32-bit write data bus.

Figure 7-2. Valid Data on Byte Lanes During a Write Transaction



In the above example, if you set all *write_strobes[]* array elements to 1 prior to executing the write transaction, automatic correction produces the following results during execution of the transaction.

	Prior to Execution		During Execution
1st data phase	write_strobes[0]=0b1111	->	write_strobes[0]=0b0010
2nd data phase	write_strobes[1]=0b1111	->	write_strobes[1]=0b1100
3rd data phase	write_strobes[2]=0b1111	->	write_strobes[2]=0b0011

If you randomly set all *write_strobes[]* array elements to 0 or 1, prior to executing the write transaction, automatic correction *only* corrects those *write_strobes[]* array elements that were previously set to 1, as shown below.

	Prior to Execution		During Execution
1st data phase	write_strobes[0]=0b1010	->	write_strobes[0]=0b0010
2nd data phase	write_strobes[1]=0b1010	->	write_strobes[1]=0b1000
3rd data phase	write_strobes[2]=0b1010	->	write_strobes[2]=0b0010

Note

To automatically generate all WSTRB signals for a write transaction, set all *write_strobes[]* array elements to 1 prior to execution of the write transaction or write data burst.

Operation Mode

By default, each read or write transaction performs a blocking operation that prevents a following transaction from starting until the current active transaction completes.

You can configure this behavior to be nonblocking by setting the *operation_mode* transaction field to the enumerate type value **_TRANSACTION_NON_BLOCKING instead of the default **_TRANSACTION_BLOCKING.

For example, in a master BFM test program, you can create a transaction by calling the *create_read_transaction()* or *create_write_transaction()* tasks, which creates a transaction record. Before executing the transaction record, you can change the *operation_mode* as follows:

```
-- * = axi| axi4
-- ** = AXI | AXI4
-- Create a write transaction to create a transaction record
create_write_transaction(1, tr_id, bfm_index, *_tr_if_0(bfm_index));

-- Change operation_mode to be nonblocking in the transaction record
set_operation_mode(**_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
                     *_tr_if_0(bfm_index));
```

In the above example, the *bfm_index* specifies the BFM.

Channel Handshake Delay

Each of the five protocol channels have *VALID and *READY handshake signals to control the rate at which information is transferred between a master and slave. The API to control these handshake signals differs between the AXI3 BFMs and AXI4 BFMs. Refer to the AXI3 BFM Handshake Delay and AXI3 BFM Delay Mode for details of the AXI3 BFM API, and AXI3 BFM Handshake Delay for details of the AXI4 BFM API.

AXI3 BFM Handshake Delay

You can configure the delay between the *VALID and *READY handshake signals for each of the five protocol channels. You can define the delay on a per phase (beat) basis for a particular transaction, measured from the positive edge of ACLK when *VALID is asserted. The delay can also be set from the completion of a previous transaction phase (*VALID and *READY both asserted).

AXI3 BFM Handshake Signal Delay Transaction Fields

There are transaction fields to configure the desired handshake delay pattern for a particular transaction phase on any of the five protocol channels. The master BFM configures the *VALID and *READY signal delays that it asserts, and the slave BFM configures the *VALID

and *READY signal delays that it asserts. Table 7-2 specifies which operational delay transaction fields are configured by the master and slave BFMs.

Table 7-2. Handshake Signal Delay Transaction Fields

Signal	Operational Transaction Field	Configuration BFM
AWVALID	address_valid_delay	Master
AWREADY	address_ready_delay	Slave
WVALID	data_valid_delay	Master
WREADY	data_ready_delay	Slave
BVALID	write_response_valid_delay	Slave
BREADY	write_response_ready_delay	Master
ARVALID	address_valid_delay	Master
ARREADY	address_ready_delay	Slave
RVALID	data_valid_delay	Slave
RREADY	data_ready_delay	Master

Note

The data channel handshake signal transaction fields (*data_valid_delay[]* and *data_ready_delay[]*) are defined as arrays so that the *VALID to *READY delay can be configured on a per data phase (beat) basis in a transaction.

AXI4 BFM Handshake Delay

The delay between the *VALID and *READY handshake signals for each of the five protocol channels is controlled in a BFM test program using *execute_*_ready()*, *get_*_ready()* and *get_*_cycle()* procedures. The *execute_*_ready()* procedures place a value onto the *READY signals, and the *get_*_ready()* procedures retrieve a value from the *READY signals. The *get_*_cycle()* procedures wait for a *VALID signal to be asserted and are used to insert a delay between the *VALID and *READY signals in the BFM test program.

For example, the master BFM test program code below inserts a specified delay between the read channel RVALID and RREADY handshake signals using the *execute_read_data_ready()* and *get_read_data_cycle()* procedures.

```
-- Set the RREADY signal to '0'.
execute_read_data_ready(0, 1, bfm_index, AXI4_PATH_6,
                           axi4_tr_if_6(bfm_index));

-- Wait for the RVALID signal to be asserted.
get_read_data_cycle(bfm_index, AXI4_PATH_6,
                        axi4_tr_if_6(bfm_index));
```

```
-- Add delay between RVALID and RREADY.
for i in  0 to 2 loop
   wait_on(AXI4_CLOCK_POSEDGE, bfm_index, AXI4_PATH_6,
           axi4_tr_if_6(bfm_index));
end loop;
execute_read_data_ready(1, 1, bfm_index, AXI4_PATH_6,
                           axi4_tr_if_6(bfm_index));
```

In this example, the *bfm_index* specifies the BFM.

AXI4 BFM *VALID Signal Delay Transaction Fields

The transaction record contains a *_*valid_delay* transaction field for each of the five protocol channels to configure the delay value prior to the assertion of the *VALID signal for the channel. The master BFM holds the delay configuration for the *VALID signals that it asserts, and the slave BFM holds the delay configuration for the *VALID signals that it asserts. Table 7-3 specifies which *_*valid_delay* fields are configured by the master and slave BFMs.

Table 7-3. Master and Slave *_valid_delay Configuration Fields

Signal	Operational Transaction Field	Configuration BFM
AWVALID	address_valid_delay	Master
WVALID	data_valid_delay	Master
BVALID	write_response_valid_delay	Slave
ARVALID	address_valid_delay	Master
RVALID	data_valid_delay	Slave

Note

In the transaction record, the data channel handshake signal transaction field (*data_valid_delay[]*) is defined as and array so that you can configure the *VALID delay on a per data phase (beat) basis in a transaction.

AXI4 BFM *READY Handshake Signal Delay Transaction Fields

The transaction record contains a *_*ready_delay* transaction field for each of the five protocol channels to store the delay value between the assertion of the *VALID and *READY handshake signals for the channel. Table 7-4 specifies the *_*ready_delay* field corresponding to the *READY signal delay.

Table 7-4. Master and Slave *_ready_delay Fields

Signal	Operational Transaction Field
AWREADY	address_ready_delay

Table 7-4. Master and Slave *_ready_delay Fields (cont.)

Signal	Operational Transaction Field
WREADY	data_ready_delay
BREADY	write_response_ready_delay
ARREADY	address_ready_delay
RREADY	data_ready_delay

Data Beat Done

There is a *data_beat_done* transaction field in each transaction, defined as an array, to indicate when each data phase (beat) has completed. Each element of the *data_beat_done* array is set to 1 when each data phase (beat) has completed in a data burst.

You call the *get_data_beat_done()* procedure in the master BFM test program to determine how many beats of a read data burst have completed by analyzing how many elements of the *data_beat_done* array have been set to 1. Similarly, you can call the *get_data_beat_done()* procedure in the slave BFM test program to analyze a write data burst.

Transaction Done

There is a *transaction_done* transaction field in each transaction that indicates when the transaction has completed.

In a BFM test program, you call the respective BFM *get_transaction_done()* procedure to investigate whether a read or write transaction has completed.

Chapter 8
VHDL AXI3 and AXI4 Master BFMs

This chapter provides information about the VHDL AXI3 and AXI4 master BFMs. Each BFM has an API that contains procedures to configure the BFM and to access the dynamic "Transaction Record" on page 33 during the life of the transaction.

Note

Due to AXI3 protocol specification changes, for some BFM procedures, you reference the AXI3 BFM by specifying AXI instead of AXI3.

Overloaded Procedure Common Arguments

The BFMs use VHDL procedure overloading, which results in the prototype having a number of prototype definitions for each procedure. Their arguments are unique to each procedure and concern the protocol or operational transaction fields for a transaction. These procedures have several common arguments, which can be optional, and include the arguments described below:

- *transaction_id* is an index number that identifies a specific transaction. Each new transaction automatically increments the index number until reaching 255, the maximum value, and then the index number automatically wraps to zero. The *transaction_id* uniquely identifies each transaction when there are a number of concurrently active transactions.
- *queue_id* is a unique identifier for each queue in a test bench. A queue is used to pass the record of a transaction between the address, data and response channels of a write transaction, and the address and data channels of a read transaction. There is a maximum of five queues available within an AXI3 BFM and eight queues available within an AXI4 BFM. Refer to "AXI3 Advanced Slave API Definition" *on page 651* "AXI4 Advanced Slave API Definition" *on page 663* for more details on the application of the *queue_id*.
- *bfm_id* is a unique identification number for each master, slave, and monitor BFM in a multiple BFM test bench.
- *path_id* is a unique identifier for each parallel process in a multiple process test bench. You must specify the *path_id* for test bench stimulus to replicate the pipelining features of a protocol in a VHDL test bench. If no pipelining is performed in the test bench stimulus (a single process), then specifying the *path_id* argument for the procedure is optional. There is a maximum of five paths available within an AXI3 BFM and eight paths available within an AXI4 BFM. Refer to "AXI3 Advanced Slave API Definition"

on page 651"AXI4 Advanced Slave API Definition" *on page 663* for more details on the application of the *path_id*.

- *tr_if* is a signal definition that passes the content of a transaction between the VHDL and SystemVerilog environments.

Master BFM Protocol Support

The AXI3 master BFM supports the AMBA AXI3 protocol with restrictions detailed in "Protocol Restrictions" on page 19. In addition to the standard protocol, it supports user sideband signals AWUSER and ARUSER.

The AXI4 master BFM supports the AMBA AXI4 protocol with restrictions detailed in "Protocol Restrictions" on page 19.

Master Timing and Events

For detailed timing diagrams of the protocol bus activity and details of the following master BFM API timing and events, refer to the relevant AMBA AXI Protocol Specification chapter.

The AMBA AXI Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the master BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

Master BFM Configuration

The master BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. It has parameters you can use to configure the widths of the address, ID and data signals, and transaction fields to configure timeout factors, slave exclusive support, setup and hold times, and so on.

You can change the address, ID and data signal widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the master BFM via a parameter port list of the master BFM component.

Table 8-1 lists the parameter names for the address, ID and data signals, and their default values.

Table 8-1. Master BFM Signal Width Parameters

Signal Width Parameter	Description
**_ADDRESS_WIDTH	Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.

Table 8-1. Master BFM Signal Width Parameters (cont.)

Signal Width Parameter	Description
**_RDATA_WIDTH	Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_WDATA_WIDTH	Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_ID_WIDTH	ID signal width in bits. This applies to the RID and WID signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
AXI4_USER_WIDTH	(AXI4) User data signal width in bits. This applies to the ARUSER, AWUSER, RUSER, WUSER and BUSER signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 8.
AXI4_REGION_MAP_SIZE	(AXI4) Region signal width in bits. This applies to the ARREGION and AWREGION signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 16.
index	Uniquely identifies a Master BFM instance. It must be set to a different value for each Master BFM in the system. Default: 0.
READ_ISSUING_ CAPABILITY	The maximum number of outstanding read transactions that can be issued from the Master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
WRITE_ISSUING_ CAPABILITY	The maximum number of outstanding write transactions that can be issued from the Master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
COMBINED_ISSUING_ CAPABILITY	The maximum number of outstanding combined read and write transactions that can be issued from the Master BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.

Table 8-1. Master BFM Signal Width Parameters (cont.)

Signal Width Parameter	Description
USE_*	(AXI4) Each protocol signal connection to the Master BFM can be enabled or disabled. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. 0 = disabled. 1 = enabled (default).

A master BFM has configuration fields that you can set via the *set_config()* function to configure timeout factors, slave exclusive support, setup and hold times, and so on. You can also get the value of a configuration field via the *get_config()* procedures. Table 8-2 describes the full list of configuration fields.

Table 8-2. Master BFM Configuration

Configuration Field	Description
Timing Variables	
**_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_MAX_TRANSACTION_TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER	(AXI3) The maximum number of write data beats that the AXI3 BFM can generate as part of write data burst of write transfer. Default: 1024.
**_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
**_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY	The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY	The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default: 10000.

Table 8-2. Master BFM Configuration (cont.)

Configuration Field	Description
**_CONFIG_MAX_LATENCY_RVALID_ ASSERTION_TO_RREADY	The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_BVALID_ ASSERTION_TO_BREADY	The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_WVALID_ ASSERTION_TO_WREADY	The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods. Default: 10000.
AXI_CONFIG_WRITE_CTRL_TO_DATA_ MINTIME	(AXI3) The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_WRITE_DELAY	(AXI3) The master BFM applies the AXI_CONFIG_WRITE_CTRL_TO_ DATA_MINTIME value set. 0 = true (default) 1 = false
AXI_CONFIG_MASTER_DEFAULT_UNDER_ RESET	(AXI3) The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true
AXI4_CONFIG_ENABLE_QOS	(AXI4) The master participates in the Quality-of-Service scheme. If a master does not participate, the AWQOS/ARQOS value used in write/read transactions must be b0000.
Master Attributes	
AXI4_CONFIG_ENABLE_RLAST	(AXI4) Configures the support for the optional RLAST signal. 0 = disabled 1 = enabled (default)
Slave Attributes	

Table 8-2. Master BFM Configuration (cont.)

Configuration Field	Description
**_CONFIG_SUPPORT_EXCLUSIVE_ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET	(AXI3) The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
**_CONFIG_SLAVE_START_ADDR	Configures the start address map for the slave.
**_CONFIG_SLAVE_END_ADDR	Configures the end address map for the slave.
**_CONFIG_READ_DATA_REORDERING_DEPTH	The slave read reordering depth. Refer to the AMBA AXI Protocol Specification for more details. Default: 1.
Error Detection	
**_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
**_CONFIG_ENABLE_ASSERTION	Individual enable/disable of assertion check in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to Master Timing and Events for details of simulator time-steps.

Master Assertions

Each master BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

AXI3 Assertion Configuration

By default, all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the *set_config()* command as the following example illustrates.

```
set_config(AXI_CONFIG_ENABLE_ALL_ASSERTIONS,0,bfm_index,
axi_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi_tr_if_0(bfm_index));

-- Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI_LOCK_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi_tr_if_0(bfm_index));
```

Note
Do not confuse the AXI_CONFIG_ENABLE_ASSERTION bit vector with the AXI_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to "AXI3 Assertions" on page 725.

AXI4 Assertion Configuration

By default, all built-in assertions are enabled in the master BFM. To globally disable them in the master BFM, use the *set_config()* command as the following example illustrates.

```
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS,0,bfm_index,
axi4_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI4_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4_tr_if_0(bfm_index));

-- Assign the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI4_LOCK_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion, follow the above code sequence and assign the assertion in the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to "AXI4 Assertions" on page 738.

VHDL Master API

This section describes the VHDL Master API.

set_config()

This nonblocking procedure sets the configuration of the master BFM.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure set_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val    : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0)|integer;
   bfm_id        : in integer;
   path_id       : in *_path_t; -- optional
   signal tr_if  : inout *_vhd_if_struct_t
);
```

Arguments

config_name (AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

(AXI4) Configuration name:
- AXI4_CONFIG_SETUP_TIME
- AXI4_CONFIG_HOLD_TIME
- AXI4_CONFIG_BURST_TIMEOUT_FACTOR
- AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI4_CONFIG_ENABLE_RLAST
- AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
- AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI4_CONFIG_ENABLE_ASSERTION
- AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI4_CONFIG_ENABLE_QOS
- AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI4_CONFIG_SLAVE_START_ADDR
- AXI4_CONFIG_SLAVE_END_ADDR

config_val — Refer to "Master BFM Configuration" on page 222 for description and valid values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

- **_PATH_0
- **_PATH_1
- **_PATH_2
- **_PATH_3
- **_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi_tr_if_0(bfm_index));
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi_tr_if_0(bfm_index));
```

AXI4 Example

```
set_config(AXI4_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi4_tr_if_0(bfm_index));
set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi4_tr_if_0(bfm_index));
```

get_config()

This nonblocking procedure gets the configuration of the master BFM.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val    : out std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0)|integer;
   bfm_id        : in integer;
   path_id       : in *_path_t; --optional
   signal tr_if  : inout *_vhd_if_struct_t
);
```

Arguments

config_name (AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

(AXI4) Configuration name:
- AXI4_CONFIG_SETUP_TIME
- AXI4_CONFIG_HOLD_TIME
- AXI4_CONFIG_BURST_TIMEOUT_FACTOR
- AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI4_CONFIG_ENABLE_RLAST
- AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
- AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI4_CONFIG_ENABLE_ASSERTION
- AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI4_CONFIG_ENABLE_QOS
- AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI4_CONFIG_SLAVE_START_ADDR
- AXI4_CONFIG_SLAVE_END_ADDR

config_val — Refer to "Master BFM Configuration" on page 222 for description and valid values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

- **_PATH_0
- **_PATH_1
- **_PATH_2
- **_PATH_3
- **_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns config_val

AXI3 Example

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
```

AXI4 Example

```
get_config(AXI4_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value,
     bfm_index, axi4_tr_if_0(bfm_index));
get_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
     axi4_tr_if_0(bfm_index));
```

create_write_transaction()

This nonblocking procedure creates a write transaction with a start address *addr* and optional *burst_length* arguments. All other transaction fields default to legal protocol values, unless previously assigned a value. It returns with the *transaction_id* argument.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure create_write_transaction
(
   addr           : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0)|integer;
   burst_length    : in integer; --optional
   transaction_id  : out integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address
burst_length	(Optional) Burst length. Default: 0.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Protocol Transaction Fields

size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;

Protocol Transaction Fields

lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
cache	(AXI3) Burst cache: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW; (AXI4) Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;
prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
id	Burst ID.
data_words	Data words array.
write_strobes	Write strobes array: Each strobe 0 or 1.

Protocol Transaction Fields	resp	Response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;
	region	(AXI4) Region identifier.
	qos	(AXI4) Quality-of-Service identifier.
	addr_user	Address channel user data.
	data_user	(AXI4) Data channel user data.
	resp_user	(AXI4) Response channel user data.
Operational Transaction Fields	gen_write_strobes	Correction of write strobes for invalid byte lanes: 0 = write_strobes passed through to protocol signals. 1 = write_strobes auto-corrected for invalid byte lanes (default).
	operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING; (default)
	delay_mode	(AXI3) Delay mode: AXI_VALID2READY; (default) AXI_TRANS2READY;
	write_data_mode	Write data mode: **_DATA_AFTER_ADDRESS; The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS; (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*
	address_valid_delay	Address channel A*VALID delay measured in ACLK cycles for this transaction (default = 0).
	data_valid_delay	Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
	write_response_read_delay	Write response channel BREADY delay measured in ACLK cycles for this transaction (default = 0).
	data_beat_done	Write data channel beat *done* flag array for this transaction.
	transaction_done	Write transaction *done* flag for this transaction.

Returns	transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221.

AXI3 Example

```
-- Create a write data burst of length 3 (4 beats) to start address 16.
-- Returns the transaction ID (tr_id) for this created transaction.
create_write_transaction(16, 3, tr_id, bfm_index,
      axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write data burst of length 3 (4 beats) to start address 16.
-- Returns the transaction ID (tr_i d) for this created transaction.
create_write_transaction(16, 3, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

create_read_transaction()

This nonblocking procedure creates a read transaction with a start address *addr* and optional *burst_length* arguments. All other transaction parameters default to legal protocol values, unless previously assigned a value. It returns with the *transaction_id* argument.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure create_read_transaction
(
   addr             : in std_logic_vector(**_MAX_BIT_SIZE-1
   downto 0)|integer;
   burst_length     : in integer; --optional
   transaction_id   : out integer;
   bfm_id           : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if     : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address
burst_length	(Optional) Burst length. Default: 0.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Protocol Transaction Fields

size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;

Protocol Transaction Fields

lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
cache	(AXI3) Burst cache: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW; (AXI4) Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;
prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
id	Burst ID.
data_words	Data words array.
resp	Response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;

Operational Transaction Fields	operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING; (default)
	delay_mode	(AXI3) Delay mode: AXI_VALID2READY; (default) AXI_TRANS2READY;
	address_valid_delay	Address channel A*VALID delay measured in ACLK cycles for this transaction (default = 0).
	data_ready_delay	Read data channel RREADY delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
	data_beat_done	Write data channel beat *done* flag array for this transaction.
	transaction_done	Read transaction *done* flag for this transaction.
Returns	transaction_id	

AXI3 Example

```
    -- Create a read data burst of length 3 (4 beats) with start address 16.
    -- Returns the transaction ID (tr_id) for this created transaction.
create_read_transaction(16, 3, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
    -- Create a read data burst of length 3 (4 beats) with start address 16.
    -- Returns the transaction ID (tr_id) for this created transaction.
    create_read_transaction(16, 3, tr_id, bfm_index,
    axi4_tr_if_0(bfm_index));
```

set_addr()

This nonblocking procedure sets the start address *addr* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_addr
(
   addr : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address of transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221. for more details

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the start address to 1 for the tr_id transaction
set_addr(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the start address to 1 for the tr_id transaction
set_addr(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_addr()

This nonblocking procedure gets the start address *addr* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_addr
(
   addr : out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address of transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns addr

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_size()

This nonblocking procedure sets the burst *size* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_size
(
   size : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the burst size to 4 bytes for the tr_id transaction
set_size (AXI_BYTES_4, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the burst size to 4 bytes for the tr_id transaction
set_size (AXI4_BYTES_4, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_size()

This nonblocking procedure gets the burst *size* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_size
(
   size : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

size	Burst size: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to for more details.
path_id	(Optional) Parallel process path identifier of type **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns size

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst size of the tr_id transaction.
get_size (size, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst size of the tr_id transaction.
get_size (size, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_burst()

This nonblocking procedure sets the *burst* type field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_burst
(
   burst: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst	Burst type: **_FIXED; **_INCR (default); **_WRAP; **_BURST_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the burst type to wrap for the tr_id transaction.
set_burst (AXI_WRAP, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the burst type to wrap for the tr_id transaction.
set_burst (AXI4_WRAP, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_burst()

This nonblocking procedure gets the *burst* type field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_burst
(
   burst: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst	Burst type: **_FIXED; **_INCR; **_WRAP; **_BURST_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns burst

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst type of the tr_id transaction.
get_burst (burst, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst type of the tr_id transaction.
get_burst (burst, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_lock()

This nonblocking procedure sets the *lock* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_lock
(
   lock : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

lock	Burst lock: **_NORMAL (default); **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the lock field to exclusive for the tr_id transaction.
set_lock(AXI_EXCLUSIVE, tr_id, bfm_index, axi_tr_if_0(bfm_index))
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the lock field to exclusive for the tr_id transaction.
set_lock(AXI4_EXCLUSIVE, tr_id, bfm_index, axi4_tr_if_0(bfm_index))
```

get_lock()

This nonblocking procedure gets the *lock* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_lock
(
   lock : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

lock	Burst lock: **_NORMAL; **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

lock

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the lock field of the tr_id transaction.
get_lock(lock, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the lock field of the tr_id transaction.
get_lock(lock, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_cache()

This nonblocking procedure sets the *cache* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_cache
(
   cache: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

cache — (AXI3) Burst cache:
- AXI_NONCACHE_NONBUF; (default)
- AXI_BUF_ONLY;
- AXI_CACHE_NOALLOC;
- AXI_CACHE_BUF_NOALLOC;
- AXI_CACHE_RSVD0;
- AXI_CACHE_RSVD1;
- AXI_CACHE_WTHROUGH_ALLOC_R_ONLY;
- AXI_CACHE_WBACK_ALLOC_R_ONLY;
- AXI_CACHE_RSVD2;
- AXI_CACHE_RSVD3;
- AXI_CACHE_WTHROUGH_ALLOC_W_ONLY;
- AXI_CACHE_WBACK_ALLOC_W_ONLY;
- AXI_CACHE_RSVD4;
- AXI_CACHE_RSVD5;
- AXI_CACHE_WTHROUGH_ALLOC_RW;
- AXI_CACHE_WBACK_ALLOC_RW;

(AXI4) Burst cache:
- AXI4_NONMODIFIABLE_NONBUF; (default)
- AXI4_BUF_ONLY;
- AXI4_CACHE_NOALLOC;
- AXI4_CACHE_2;
- AXI4_CACHE_3;
- AXI4_CACHE_RSVD4;
- AXI4_CACHE_RSVD5;
- AXI4_CACHE_6;
- AXI4_CACHE_7;
- AXI4_CACHE_RSVD8;
- AXI4_CACHE_RSVD9;
- AXI4_CACHE_10;
- AXI4_CACHE_11;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_14;
- AXI4_CACHE_15;

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the cache field to bufferable only for the tr_id transaction.
set_cache(AXI_BUF_ONLY, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the tran4saction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the cache field to bufferable only for the tr_id transaction.
set_cache(AXI4_BUF_ONLY, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_cache()

This nonblocking procedure gets the *cache* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_cache
(
   cache: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

cache — (AXI3) Burst cache:
- AXI_NONCACHE_NONBUF; (default)
- AXI_BUF_ONLY;
- AXI_CACHE_NOALLOC;
- AXI_CACHE_BUF_NOALLOC;
- AXI_CACHE_RSVD0;
- AXI_CACHE_RSVD1;
- AXI_CACHE_WTHROUGH_ALLOC_R_ONLY;
- AXI_CACHE_WBACK_ALLOC_R_ONLY;
- AXI_CACHE_RSVD2;
- AXI_CACHE_RSVD3;
- AXI_CACHE_WTHROUGH_ALLOC_W_ONLY;
- AXI_CACHE_WBACK_ALLOC_W_ONLY;
- AXI_CACHE_RSVD4;
- AXI_CACHE_RSVD5;
- AXI_CACHE_WTHROUGH_ALLOC_RW;
- AXI_CACHE_WBACK_ALLOC_RW;

(AXI4) Burst cache:
- AXI4_NONMODIFIABLE_NONBUF; (default)
- AXI4_BUF_ONLY;
- AXI4_CACHE_NOALLOC;
- AXI4_CACHE_2;
- AXI4_CACHE_3;
- AXI4_CACHE_RSVD4;
- AXI4_CACHE_RSVD5;
- AXI4_CACHE_6;
- AXI4_CACHE_7;
- AXI4_CACHE_RSVD8;
- AXI4_CACHE_RSVD9;
- AXI4_CACHE_10;
- AXI4_CACHE_11;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_14;
- AXI4_CACHE_15;

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

	path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
	tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
Returns	cache	

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the cache field of the tr_id transaction.
get_cache(cache, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the cache field of the tr_id transaction.
get_cache(cache, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_prot()

This nonblocking procedure sets the protection *prot* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_prot
(
   prot: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

prot	Protection: **_NORM_SEC_DATA (default); **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the protection field to a normal, secure, instruction access
-- for the tr_id transaction.
set_prot(AXI_NORM_SEC_INST, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the protection field to a normal, secure, instruction access
-- for the tr_id transaction.
set_prot(AXI4_NORM_SEC_INST, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_prot()

This nonblocking procedure gets the protection *prot* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_prot
(
   prot: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

prot	Protection: **_NORM_SEC_DATA; **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

prot

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_id()

This nonblocking procedure sets the *id* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_id
(
   id: in integer;
   transaction_id  : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0) | integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

id	Burst ID
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the id field to 2 for the tr_id transaction.
set_id(2, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the id field to 2 for the tr_id transaction.
set_id(2, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_id()

This nonblocking procedure gets the *id* field for a transaction, which uniquely identifies the transaction defined by the *transaction_id* field and previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_id
(
   id: out integer;
   transaction_id  : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0) | integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

id	Burst ID
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

id

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_burst_length()

This nonblocking procedure sets the *burst_length* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Note

The *burst_length* field is the value that appears on the AWLEN and the ARLEN protocol signals. The number of data phases (beats) in a data burst is therefore *burst_length* + 1.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_burst_length
(
  burst_length : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
  integer;
  transaction_id  : in integer;
  bfm_id : in integer;
  path_id : in *_path_t; --optional
  signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst_length	Burst length. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the burst length field to 2 (3 beats) for the tr_id transaction.
set_burst_length(2, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the tran4saction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the burst length field to 2 (3 beats) for the tr_id transaction.
set_burst_length(2, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_burst_length()

This nonblocking procedure gets the *burst_length* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Note

The *burst_length* field is the value that appears on the AWLEN and the ARLEN protocol signals. The number of data phases (beats) in a data burst is the *burst_length* + 1.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_burst_length
(
 burst_length : out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0)
 | integer;
 transaction_id  : in integer;
 bfm_id : in integer;
 path_id : in *_path_t; --optional
 signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst_length	Burst length.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

burst_length

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst length field of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst length field of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_words()

This nonblocking procedure sets a *data_words* field array element for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_data_words
(
   data_words: in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_words	Data words array.
index	(Optional) Array element index number for data_words.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the data_words field to 2 for the first write data phase (beat)
-- for the tr_id transaction.
set_data_words(2, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the data_words field to 3 for the second write data phase (beat)
-- for the tr_id transaction.
set_data_words(3, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the data_words field to 2 for the first write data phase (beat)
-- for the tr_id transaction.
set_data_words(2, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the data_words field to 3 for the second write data phase (beat)
-- for the tr_id transaction.
set_data_words(3, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_data_words()

This nonblocking procedure gets a *data_words* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_data_words
(
   data_words: out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_words	Data words array.
index	(Optional) Array element index number for data_words.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns data_words

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the data_words field for the first data phase (beat)
-- of the tr_id transaction.
get_data_words(data, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the data_words field for the first data phase (beat)
-- of the tr_id transaction.
get_data_words(data, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the data_words field for the second data phase (beat)
-- of the tr_id transaction.
get_data_words(data, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_write_strobes()

This nonblocking procedure sets the *write_strobes* field array elements for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_write_strobes
(
   write_strobes : in std_logic_vector (**_MAX_BIT_SIZE-1 downto
   0) | integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_strobes	Write strobes array.
index	(Optional) Array element index number for write_strobes.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the write_strobes field to for the first data phase (beat)
-- for the tr_id transaction.
set_write_strobes(2, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the write_strobes field to 12 for the second data phase (beat)
-- for the tr_id transaction.
set_write_strobes(12, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write_strobes field to for the first data phase (beat)
-- for the tr_id transaction.
set_write_strobes(2, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write_strobes field to 12 for the second data phase (beat)
-- for the tr_id transaction.
set_write_strobes(12, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_strobes()

This nonblocking procedure gets a *write_strobes* field array element for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_write_strobes
(
 write_strobes : out std_logic_vector (**_MAX_BIT_SIZE-1 downto 0)
 | integer;
 index : in integer; --optional
 transaction_id  : in integer;
 bfm_id : in integer;
 path_id : in *_path_t; --optional
 signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_strobes	Write strobes array.
index	(Optional) Array element index number fro write_strobes.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

write_strobes

AXI3 Example

```
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the write_strobes field for the first data phase (beat)
-- of the tr_id transaction.
get_write_strobes(write_strobe, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the write_strobes field for the second data phase (beat)
-- of the tr_id transaction.
get_write_strobes(write_strobe, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write_strobes field for the first data phase (beat)
-- of the tr_id transaction.
get_write_strobes(write_strobe, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the write_strobes field for the second data phase (beat)
-- of the tr_id transaction.
get_write_strobes(write_strobe, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_resp()

This nonblocking procedure sets a response *resp* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_resp
(
   resp: in std_logic_vector (**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

resp	Transaction response array: **_OKAY = 0; **_EXOKAY = 1; **_SLVERR = 2; **_DECERR = 3;
index	(Optional) Array element index number for resp.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

Note

You would not normally use this procedure in a master test program.

get_resp()

This nonblocking procedure gets a response *resp* field array element for a transaction that is identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_resp
(
   resp: out std_logic_vector (**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

resp — Transaction response array:

**_OKAY = 0;
**_EXOKAY = 1;
**_SLVERR = 2;
**_DECERR = 3;

index — (Optional) Array element index number for resp.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

resp

AXI3 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the response field for the first data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the response field for the second data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 1.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the response field for the first data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the response field for the second data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_addr_user()

This nonblocking procedure sets the user data *addr_user* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_addr_user
(
   addr_user : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr_user	User data in address phase.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the user data to 1 for tr_id transaction.
set_addr_user(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the user data to 1 for tr_id transaction.
set_addr_user(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_addr_user()

This nonblocking procedure gets the user data *addr_user* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* or *create_read_transaction()* procedures.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_addr_user
(
   addr_user : out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr_user	User data in the address phase.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns addr_user

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel user data of the tr_id transaction.
get_addr_user(user_data, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel user data of the tr_id transaction.
get_addr_user(user_data, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_read_or_write()

This nonblocking procedure sets the *read_or_write* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_read_or_write
(
   read_or_write: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

read_or_write	Read or write transaction: **_TRANS_READ = 0 **_TRANS_WRITE = 1
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a master test program.

get_read_or_write()

This nonblocking procedure gets the *read_or_write* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4 |
-- ** = AXI | AXI4
get_read_or_write
(
   read_or_write: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

read_or_write	Read or write transaction: **_TRANS_READ = 0 **_TRANS_WRITE = 1
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

read_or_write

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read_or_write field of the tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read_or_write field of the tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_gen_write_strobes()

This nonblocking procedure sets the *gen_write_strobes* field for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_gen_write_strobes
(
   gen_write_strobes: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

gen_write_strobes	Correction of write strobes for invalid byte lanes: 0 = *write_strobes* passed through to protocol signals. 1 = *write_strobes* auto-corrected for invalid byte lanes (default).
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Disable the auto correction of the write strobes for the
-- tr_id transaction.
set_gen_write_strobes(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Disable the auto correction of the write strobes for the
-- tr_id transaction.
set_gen_write_strobes(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_gen_write_strobes()

This nonblocking procedure gets the *gen_write_strobes* field for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_gen_write_strobes
(
   gen_write_strobes: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

gen_write_strobes	Correct write strobes flag: 0 = write_strobes passed through to protocol signals. 1 = write_strobes auto-corrected for invalid byte lanes.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns gen_write_strobes

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_operation_mode()

This nonblocking procedure sets the *operation_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_operation_mode
(
   operation_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING (default);
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the operation mode field to nonblocking for tr_id transaction.
set_operation_mode(AXI_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the operation mode field to nonblocking for tr_id transaction.
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

get_operation_mode()

This nonblocking procedure gets the *operation_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_operation_mode
(
   operation_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns operation_mode

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the operation mode field of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the operation mode field of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_delay_mode()

This AXI3 nonblocking procedure sets the *delay_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
set_delay_mode
(
   delay_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

delay_mode	Delay mode: AXI_VALID2READY (default); AXI_TRANS2READY;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the delay mode of the *VALID to *READY handshake for the
-- tr_id transaction.
set_delay_mode(AXI_VALID2READY, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

get_delay_mode()

This AXI3 nonblocking procedure gets the *delay_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
get_delay_mode
(
   delay_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

delay_mode	Delay mode: AXI_VALID2READY; AXI_TRANS2READY;
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

delay_mode

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the delay mode of the *VALID to *READY handshake of the
-- tr_id transaction.
get_delay_mode(delay_mode, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

set_write_data_mode()

This nonblocking procedure sets the *write_data_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_write_data_mode
(
   write_data_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_data_mode — Write data mode:

**_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation.

**_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the write data mode field of the address and data phases for the
-- tr_id transaction
set_write_data_mode(AXI_DATA_WITH_ADDRESS, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write data mode field of the address and data phases for the
-- tr_id transaction
set_write_data_mode(AXI4_DATA_WITH_ADDRESS, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

get_write_data_mode()

This nonblocking procedure gets the *write_data_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_write_data_mode
(
   write_data_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_data_mode — Write data mode:

**_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation.

**_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

write_data_mode

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data mode field of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data mode field of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_address_valid_delay()

This nonblocking procedure sets the *address_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_address_valid_delay
(
   address_valid_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_valid_delay	Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the address channel *VALID delay to 3 clock cycles
-- for the tr_id transaction.
set_address_valid_delay(3, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the address channel *VALID delay to 3 clock cycles
-- for the tr_id transaction.
set_address_valid_delay(3, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_address_valid_delay()

This nonblocking procedure gets the *address_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_address_valid_delay
(
   address_valid_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_valid_delay	Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns address_valid_delay

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write address channel AWVALID delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write address channel AWVALID delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_address_ready_delay()

This AXI3 nonblocking procedure sets the *address_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
set_address_ready_delay
(
   address_ready_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in _path_t; --optional
   signal tr_if : inout _vhd_if_struct_t
);
```

Arguments

address_ready_delay	Address channel A*READY delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a master test program.

get_address_ready_delay()

This nonblocking procedure gets the *address_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_address_ready_delay
(
   address_ready_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_ready_delay	Address channel A*READY delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns address_ready_delay

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_valid_delay()

This nonblocking procedure sets the *data_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_data_valid_delay
(
   data_valid_delay: in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_valid_delay	Write data channel WVALID delay measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element index number for data_valid_delay.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the write channel WVALID delay to 3 ACLK cycles for the first data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(3, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the write channel WVALID delay to 2 ACLK cycles for the second data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(2, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write channel WVALID delay to 3 ACLK cycles for the first data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(3, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write channel WVALID delay to 2 ACLK cycles for the second data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(2, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_data_valid_delay()

This nonblocking procedure gets the *data_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_data_valid_delay
(
   data_valid_delay: out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_valid_delay	Data channel array to store *VALID delays measured in ACLK cycles for this transaction.
index	(Optional) Array element index number for data_valid_delay.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

data_valid_delay

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read channel RVALID delay for the first data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the read channel RVALID delay for the second data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read channel RVALID delay for the first data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the read channel RVALID delay for the second data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

get_data_ready_delay()

This nonblocking procedure gets the *data_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_data_ready_delay
(
   data_ready_delay: out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_ready_delay	Read data channel RREADY delay measured in ACLK cycles for this transaction.
index	(Optional) Array element index number for data_ready_delay.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

data_ready_delay

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the write data channel WREADY delay the first
-- dataphase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the write data channel WREADY delay for the second
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write data channel WREADY delay the first
-- dataphase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the write data channel WREADY delay for the second
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_write_response_valid_delay()

This nonblocking procedure sets the *write_response_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_write_response_valid_delay
(
   write_response_valid_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_valid_delay	Write data channel BVALID delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a master test program.

get_write_response_valid_delay()

This nonblocking procedure gets the *write_response_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_write_response_valid_delay
(
   write_response_valid_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_valid_delay	Write data channel BVALID delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

write_response_valid_delay

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id,
bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id,
bfm_index, axi4_tr_if_0(bfm_index));
```

set_write_response_ready_delay()

This AXI3 nonblocking procedure sets the *write_response_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
set_write_response_ready_delay
(
   write_response_ready_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the write response channel BREADY delay to 3 ACLK cycles
-- of the tr_id transaction.
set_write_response_ready_delay(3, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

get_write_response_ready_delay()

This nonblocking procedure gets the *write_response_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_write_response_ready_delay
(
   write_response_ready_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

write_response_ready_delay

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_beat_done()

This nonblocking procedure sets the *data_beat_done* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_data_beat_done
(
   data_beat_done : in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   ath_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_beat_done	Read data channel phase (beat) *done* array for this transaction.
index	(Optional) Array element index number for data_beat_done.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Set the read data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Set the read data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Set the read data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Set the read data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_data_beat_done()

This nonblocking procedure gets the *data_beat_done* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_data_beat_done
(
   data_beat_done : out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_beat_done	Data channel phase (beat) *done* array for this transaction
index	(Optional) Array element index number for data_beat_done.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

data_beat_done

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

....

-- Get the write data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

....

-- Get the write data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_transaction_done()

This nonblocking procedure sets the *transaction_done* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_transaction_done
(
   transaction_done : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Set the read transaction_done flag of the tr_id transaction.
set_transaction_done(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Set the read transaction_done flag of the tr_id transaction.
set_transaction_done(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_transaction_done()

This nonblocking procedure gets the *transaction_done* field for a transaction that is uniquely identified by the *transaction_id* field previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_transaction_done
(
   transaction_done : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns transaction_done

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

execute_transaction()

This procedure executes a master transaction that is uniquely identified by the *transaction_id* argument, previously created with either the *create_write_transaction()* or *create_read_transaction()* procedure. A transaction can be blocking (default) or nonblocking, based on the setting of the transaction *operation_mode* field.

The results of *execute_transaction()* for write transactions varies based on how write transaction fields are set. If the transaction *gen_write_strobes* field is set, *execute_transaction ()* automatically corrects any previously set *write_strobes* field array elements. However, if the *gen_write_strobes* field is not set, then any previously assigned *write_strobes* field array elements will be passed onto the WSTRB protocol signals, which can result in a protocol violation if not correctly set. Refer to "Automatic Correction of Byte Lane Strobes" on page 214 for more details.If the *write_data_mode* field for a write transaction is set to *_DATA_WITH_ADDRESS, *execute_transaction()* calls the *execute_write_addr_phase()* and *execute_write_data_burst()* procedures simultaneously; otherwise, *execute_write_data_burst()* is called after *execute_write_addr_phase()* so that the write data burst occurs after the write address phase (default). It then calls the *get_write_response_phase()* procedure to complete the write transaction.

For a read transaction, *execute_transaction ()* calls the *execute_read_addr_phase()* procedure followed by the *get_read_data_burst()* procedure to complete the read transaction.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure execute_transaction
(
   transaction_id : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the ID to 1 for this transaction
set_id(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Execute the tr_id transaction.
execute_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the ID to 1 for this transaction
set_id(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Execute the tr_id transaction.
execute_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

execute_write_addr_phase()

This procedure executes a master write address phase uniquely identified by the *transaction_id* argument previously created by the *create_write_transaction()* procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

It sets the AWVALID protocol signal at the appropriate time defined by the transaction record *address_valid_delay* field.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure execute_write_addr_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the ID to 1 for this transaction
set_id(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Execute the write address phase for the tr_id transaction.
execute_write_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the ID to 1 for this transaction
set_id(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Execute the write address phase for the tr_id transaction.
execute_write_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

execute_read_addr_phase()

This procedure executes a master read address phase uniquely identified by the *transaction_id* argument previously created by the *create_read_transaction()* procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

It sets the ARVALID protocol signal at the appropriate time defined by the transaction record *address_valid_delay* field.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure execute_read_addr_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the ID to 1 for this transaction
set_id(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Execute the read address phase for the tr_id transaction.
execute_read_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the ID to 1 for this transaction
set_id(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Execute the read address phase for the tr_id transaction.
execute_read_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

execute_write_data_burst()

This procedure executes a write data burst that is uniquely identified by the *transaction_id* argument previously created by the *create_write_transaction()* procedure. This burst can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

If the transaction record *gen_write_strobes* field is set, it automatically corrects any previously set *write_strobes* field array elements. If the *gen_write_strobes* field is not set, then any previously assigned *write_strobes* field array elements are passed through onto the WSTRB protocol signals, which can result in a protocol violation if not correctly set. Refer to "Automatic Correction of Byte Lane Strobes" on page 214 for more details.

It calls the *execute_write_data_phase()* procedure for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure execute_write_data_burst
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Execute the write data burst for the tr_id transaction.
execute_write_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Execute the write data burst for the tr_id transaction.
execute_write_data_burst(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

execute_write_data_phase()

This procedure executes a write data phase that is uniquely identified by the *transaction_id* argument and previously created by the *create_write_transaction()* procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

The *execute_write_data_phase()* sets the WVALID protocol signal at the appropriate time defined by the transaction record field *data_valid_delay* array *index* element and sets the *data_beat_done* array *index* element to 1 when the phase completes.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure execute_write_data_phase
(
   transaction_id  : in integer;
   index : in integer; --optional
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
index	(Optional) Data phase (beat) number.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Execute the write data phase for the first beat of the
-- tr_id transaction.
execute_write_data_phase(tr_id, 0, bfm_index, axi_tr_if_0(bfm_index));

-- Execute the write data phase for the second beat of the
-- tr_id transaction.
execute_write_data_phase(tr_id, 1, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Execute the write data phase for the first beat of the
-- tr_id transaction.
execute_write_data_phase(tr_id, 0, bfm_index, axi4_tr_if_0(bfm_index));

-- Execute the write data phase for the second beat of the
-- tr_id transaction.
execute_write_data_phase(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_data_burst()

This blocking procedure gets a read data burst uniquely identified by the *transaction_id* argument previously created by the *create_read_transaction()* procedure.

It calls the *get_read_data_phase()* procedure for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_read_data_burst
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read data burst for the tr_id transaction.
get_read_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read data burst for the tr_id transaction.
get_read_data_burst(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_data_phase()

This blocking procedure gets a read data phase that is uniquely identified by the *transaction_id* argument previously created by the *create_read_transaction()* procedure. It sets the *data_beat_done* array *index* element field to 1 when the phase completes. If this is the last phase (beat) of the burst, then it sets the *transaction_done* field to 1 to indicate the whole read transaction is complete.

Note

For AXI3 the *get_read_data_phase()* also sets the RREADY protocol signal at the appropriate time defined by the transaction record *data_ready_delay* field when the phase completes.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_read_data_phase
(
   transaction_id  : in integer;
   index : in integer; --optional
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
index	(Optional) Data phase (beat) number.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read data phase for the first beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 0, bfm_index, axi_tr_if_0(bfm_index));

-- Get the read data phase for the second beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 1, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a read transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_read_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read data phase for the first beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 0, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the read data phase for the second beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_response_phase()

This blocking procedure gets a write response phase that is uniquely identified by the *transaction_id* argument previously created by the *create_write_transaction()* procedure. It sets the *transaction_done* field to 1 when the transaction completes to indicate the whole transaction is complete.

Note

For AXI3 the *get_write_response_phase()* also sets the BREADY protocol signal at the appropriate time defined by the transaction record *write_response_ready_delay* field when the phase completes.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure get_write_response_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response phase for the tr_id transaction.
get_write_response_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response phase for the tr_id transaction.
get_write_response_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_addr_ready()

This blocking AXI4 procedure returns the value of the read address channel ARREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_read_addr_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the ARREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_read_addr_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the ARREADY signal value
bfm.get_read_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_data_cycle()

This blocking AXI4 procedure waits until the read data channel RVALID signal has been asserted.

Prototype

```
procedure get_read_data_cycle
(
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 BFM

Note

The *get_read_data_cycle()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Wait for the RVALID signal to be asserted.
bfm.get_read_data_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```

execute_read_data_ready()

This AXI4 procedure executes a read data ready by placing the *ready* argument value onto the RREADY signal. It will block (default) for one ACLK period.

Prototype

```
procedure execute_read_data_ready
(
   ready : in integer
   non_blocking_mode : in integer; --optional
   bfm_id           : in integer;
   path_id           : in axi4_path_t; --optional
   signal tr_if     : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value to be placed onto the RREADY signal
non_blocking_mode	(Optional) Nonblocking mode: 0 = Nonblocking 1 = Blocking (default)
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_0 AXI4_PATH_1 AXI4_PATH_2 AXI4_PATH_3 AXI4_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

AXI3 BFM

Note

The *execute_read_data_ready()* task is not available in the AXI3 BFM. Use the *get_read_data_phase()* task along with the transaction record *data_ready_delay* field.

AXI4 Example

```
-- Set the RREADY signal to 1 and block for 1 ACLK cycle
execute_read_data_ready(1, 1, index, AXI4_PATH_6, axi4_tr_if_6(index));
```

get_write_addr_ready()

This blocking AXI4 procedure returns the value of the write address channel AWREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_write_addr_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the AWREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_write_addr_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the AWREADY signal value
bfm.get_write_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_data_ready()

This blocking AXI4 procedure returns the value of the write data channel WREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_write_data_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the WREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_write_data_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the WREADY signal value
bfm.get_write_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_response_cycle()

This blocking AXI4 procedure waits until the write response channel BVALID signal has been asserted.

Prototype

```
procedure get_write_response_cycle
(
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

AXI3 BFM

Note

The *get_write_response_cycle()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Wait for the BVALID signal to be asserted.
bfm.get_write_response_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```

execute_write_resp_ready()

This AXI4 procedure executes a write response ready by placing the *ready* argument value onto the BREADY signal. It will block for one ACLK period.

Prototype

```
 procedure execute_write_resp_ready
(
   ready : in integer;
   non_blocking_mode : in integer; --optional
   bfm_id          : in integer;
   path_id          : in axi4_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value to be placed onto the BREADY signal
non_blocking_mode	(Optional) Nonblocking mode: 0 = Nonblocking 1 = Blocking (default)
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_0 AXI4_PATH_1 AXI4_PATH_2 AXI4_PATH_3 AXI4_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 BFM

Note

The *execute_write_resp_ready()* task is not available in the AXI3 BFM. Use the *get_write_response_phase()* task along with the transaction record *write_response_ready_delay* field.

AXI4 Example

```
-- Set the BREADY signal to 1 and block for 1 ACLK cycle
execute_write_resp_ready(1, 1, index, AXI4_PATH_5, axi4_tr_if_5(index));
```

push_transaction_id()

This nonblocking procedure pushes a transaction ID into the back of a queue. The transaction is uniquely identified by the *transaction_id* argument previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure. The queue is identified by the *queue_id* argument.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure push_transaction_id
(
   transaction_id  : in integer;
   queue_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

queue_id — Queue identifier:

**_QUEUE_ID_0
**_QUEUE_ID_1
**_QUEUE_ID_2
**_QUEUE_ID_3
**_QUEUE_ID_4
AXI4_QUEUE_ID_5
AXI4_QUEUE_ID_6
AXI4_QUEUE_ID_7

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI_QUEUE_ID_1, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
```

pop_transaction_id()

This nonblocking (unless queue is empty) procedure pops a transaction ID from the front of a queue. The transaction is uniquely identified by the *transaction_id* argument previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure. The queue is identified by the *queue_id* argument.

If the queue is empty, then it will block until an entry becomes available.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure pop_transaction_id
(
   transaction_id  : in integer;
   queue_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

queue_id — Queue identifier:

- **_QUEUE_ID_0
- **_QUEUE_ID_1
- **_QUEUE_ID_2
- **_QUEUE_ID_3
- **_QUEUE_ID_4
- AXI4_QUEUE_ID_5
- AXI4_QUEUE_ID_6
- AXI4_QUEUE_ID_7

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

- **_PATH_0
- **_PATH_1
- **_PATH_2
- **_PATH_3
- **_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI_QUEUE_ID_1, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
```

print()

This nonblocking procedure prints a transaction record that is uniquely identified by the *transaction_id* argument previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure print
(
   transaction_id  : in integer;
   print_delays : in integer; --optional
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

print_delays — (Optional) Print delay values flag:

0 = do not print the delay values (default).
1 = print the delay values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```

destruct_transaction()

This blocking procedure removes a transaction record for cleanup purposes and memory management that is uniquely identified by the *transaction_id* argument previously created by either the *create_write_transaction()* or *create_read_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure destruct_transaction
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a write transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_write_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

wait_on()

This blocking task waits for an event(s) on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure wait_on
(
   phase           : in integer;
   count: in integer; --optional
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

phase	Wait for: **_CLOCK_POSEDGE **_CLOCK_NEGEDGE **_CLOCK_ANYEDGE **_CLOCK_0_TO_1 **_CLOCK_1_TO_0 **_RESET_POSEDGE **_RESET_NEGEDGE **_RESET_ANYEDGE **_RESET_0_TO_1 **_RESET_1_TO_0
count	(Optional) Wait for a number of events to occur set by *count*. (default = 1)
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
wait_on(AXI_RESET_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
wait_on(AXI_CLOCK_POSEDGE, 10, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
wait_on(AXI4_RESET_POSEDGE, bfm_index, axi4_tr_if_0(bfm_index));
wait_on(AXI4_CLOCK_POSEDGE, 10, bfm_index, axi4_tr_if_0(bfm_index));
```

Chapter 9
VHDL AXI3 and AXI4 Slave BFMs

This chapter provides information about the VHDL AXI3 and AXI4 slave BFMs. Each BFM has an API that contains procedures to configure the BFM and to access the “Transaction Record” on page 33 during the lifetime of the transaction.

Note

Due to AXI3 protocol specification changes, for some BFM tasks, you reference the AXI3 BFM by specifying AXI instead of AXI3.

Slave BFM Protocol Support

The AXI3 Slave BFM implements the AMBA AXI3 protocol with restrictions detailed in “Protocol Restrictions” on page 19. In addition to the standard protocol, it supports user sideband signals AWUSER and ARUSER.

The AXI4 slave BFM supports the AMBA AXI4 protocol with restrictions detailed in “Protocol Restrictions” on page 19.

Slave Timing and Events

For detailed timing diagrams of the protocol bus activity, refer to the relevant AMBA AXI Protocol Specification chapter, which you can use to reference details of the following slave BFM API timing and events.

The specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the slave BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

Slave BFM Configuration

The slave BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. The BFM has parameters that you can use to configure the widths of the address, ID and data signals and transaction fields to configure timeout factors, slave exclusive support, setup and hold times, and so on.

You can change the address, ID and data signals widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the slave BFM via a parameter port list of the slave BFM component.

Table 9-1 lists the parameter names for the address, ID and data signals, and their default values.

Note

See "Running the Qsys Tool" on page 676 for details of the Qsys Parameter Editor.

Table 9-1. Slave BFM Signal Width Parameters

Signal Width Parameter	Description
**_ADDRESS_WIDTH	Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
**_RDATA_WIDTH	Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_WDATA_WIDTH	Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_ID_WIDTH	ID signal width in bits. This applies to the RID and WID signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
AXI4_USER_WIDTH	(AXI4) User data signal width in bits. This applies to the ARUSER, AWUSER, RUSER, WUSER and BUSER signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 8.
AXI4_REGION_MAP_SIZE	(AXI4) Region signal width in bits. This applies to the ARREGION and AWREGION signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 16.
index	Uniquely identifies a slave BFM instance. It must be set to a different value for each slave BFM in the system. Default: 0.
READ_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding read transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See "Running the Qsys Tool" on page 676. for details. Default: 16.

Table 9-1. Slave BFM Signal Width Parameters (cont.)

WRITE_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
COMBINED_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding combined read and write transactions that can be accepted by the slave BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
USE_*	(AXI4) Each protocol signal connection to the slave BFM can be enabled or disabled. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. 0 = disabled. 1 = enabled (default).

A slave BFM has configuration fields that you can set via the *set_config()* function to configure timeout factors, slave exclusive support, setup and hold times, and so on. You can also get the value of a configuration field via the *get_config()* procedures. Table 9-2 describes the full list of configuration fields.

Table 9-2. Slave BFM Configuration

Configuration Field	Description
Timing Variables	
**_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_MAX_TRANSACTION_ TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_DATA_ TRANSFER	(AXI3) The maximum number of write data beats that the AXI3 BFM can generate as part of write data burst of write transfer. Default: 1024.
**_CONFIG_BURST_TIMEOUT_ FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.

Table 9-2. Slave BFM Configuration (cont.)

Configuration Field	Description
**_CONFIG_MAX_LATENCY_ AWVALID_ASSERTION_TO_AWREADY	The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_ ARVALID_ASSERTION_TO_ARREADY	The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_RVALID_ ASSERTION_TO_RREADY	The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_BVALID_ ASSERTION_TO_BREADY	The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods (default 10000).
**_CONFIG_MAX_LATENCY_WVALID_ ASSERTION_TO_WREADY	The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods (default 10000).
AXI_CONFIG_WRITE_CTRL_TO_DATA_ MINTIME	(AXI3) The minimum delay from the start of a write control (address) phase to the start of a write data phase in clock cycles. Default: 1.
AXI_CONFIG_MASTER_WRITE_DELAY	(AXI3) The master BFM applies the AXI_CONFIG_WRITE_CTRL_TO_ DATA_MINTIME value set. 0 = true (default) 1 = false
AXI_CONFIG_MASTER_DEFAULT_ UNDER_RESET	(AXI3) The master BFM drives the ARVALID, AWVALID and WVALID signals low during reset: 0 = false (default) 1 = true
AXI4_CONFIG_ENABLE_QOS	(AXI4) The master participates in the Quality-of-Service scheme. If a master does not participate, the AWQOS/ARQOS value used in write/read transactions must be b0000.

Table 9-2. Slave BFM Configuration (cont.)

Configuration Field	Description
Master Attributes	
AXI4_CONFIG_ENABLE_RLAST	(AXI4) Configures the support for the optional RLAST signal. 0 = disabled 1 = enabled (default)
Slave Attributes	
**_CONFIG_SUPPORT_EXCLUSIVE_ ACCESS	Configures the support for an exclusive slave. If enabled the BFM will expect an EXOKAY response to a successful exclusive transaction. If disabled the BFM will expect an OKAY response to an exclusive transaction. Refer to the AMBA AXI Protocol Specification for more details. 0 = disabled 1 = enabled (default)
AXI_CONFIG_SLAVE_DEFAULT_UNDER_ RESET	(AXI3) The slave BFM drives the BVALID and RVALID signals low during reset. Refer to the AMBA AXI Protocol Specification for more details. 0 = false (default) 1 = true
**_CONFIG_SLAVE_START_ADDR	Configures the start address map for the slave.
**_CONFIG_SLAVE_END_ADDR	Configures the end address map for the slave.
**_CONFIG_READ_DATA_REORDERING_ DEPTH	The slave read reordering depth. Refer to the AMBA AXI Protocol Specification for more details. Default: 1.
AXI4_CONFIG_MAX_OUTSTANDING_WR	(AXI4) Configures the maximum number of outstanding write requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal AWREADY=0b0 if this value is exceeded. Default = 0.

Table 9-2. Slave BFM Configuration (cont.)

Configuration Field	Description
AXI4_CONFIG_MAX_OUTSTANDING_RD	(AXI4) Configures the maximum number of outstanding read requests from the master that can be processed by the slave. The slave back-pressures the master by setting the signal ARREADY=0b0 if this value is exceeded. Default = 0.
AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE	(AXI4) Holds the number of outstanding write phases from the master that can be processed by the slave. Default = 0.
AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE	(AXI4) Holds the number of outstanding read phases to the master that can be processed by the slave. Default = 0.
Error Detection	
**_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
**_CONFIG_ENABLE_ASSERTION	Individual enable/disable of assertion check in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to Slave Timing and Events for details of simulator time-steps.

Slave Assertions

The slave BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

AXI3 Assertion Configuration

By default, all built-in assertions are enabled in the slave BFM. To globally disable them in the slave BFM, use the *set_config()* command as the following example illustrates.

```
set_config(AXI_CONFIG_ENABLE_ALL_ASSERTIONS,0,bfm_index,
axi_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi_tr_if_0(bfm_index));

-- Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI_LOCK_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI_CONFIG_ENABLE_ASSERTION bit vector with the AXI_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion, following the above code sequence, assign the assertion in the AXI_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions. refer to “AXI3 Assertions” on page 725.

AXI4 Assertion Configuration

By default, all built-in assertions are enabled in the slave BFM. To globally disable them in the slave BFM, use the *set_config()* command as the following example illustrates.

```
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS,0,bfm_index,
axi4_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI4_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4_tr_if_0(bfm_index));

-- Assign the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI4_AWADDR_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_AWADDR_CHANGED_BEFORE_AWREADY assertion, following the above code sequence, assign the assertion in the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to "AXI4 Assertions" on page 738.

VHDL Slave API

This section describes the VHDL Slave API.

set_config()

This nonblocking procedure sets the configuration of the slave BFM.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure set_config
(
 config_name   : in std_logic_vector(7 downto 0);
 config_val    : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0)|
 integer;
  bfm_id        : in integer;
  path_id       : in *_path_t; --optional
  signal tr_if  : inout *_vhd_if_struct_t
);
```

Arguments

config_name — (AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS
- AXI_CONFIG_MAX_OUTSTANDING_WR
- AXI_CONFIG_MAX_OUTSTANDING_RD

(AXI4) Configuration name:
- AXI4_CONFIG_SETUP_TIME
- AXI4_CONFIG_HOLD_TIME
- AXI4_CONFIG_BURST_TIMEOUT_FACTOR
- AXI4_CONFIG_ENABLE_RLAST
- AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
- AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI4_CONFIG_ENABLE_ASSERTION
- AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI4_CONFIG_ENABLE_QOS
- AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI4_CONFIG_SLAVE_START_ADDR
- AXI4_CONFIG_SLAVE_END_ADDR
- AXI4_CONFIG_MAX_OUTSTANDING_WR
- AXI4_CONFIG_MAX_OUTSTANDING_RD
- AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE
- AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE

config_val — Refer to “Slave BFM Configuration” on page 367 for description and valid values.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

- **_PATH_0
- **_PATH_1
- **_PATH_2
- **_PATH_3
- **_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

AXI3 Example

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi_tr_if_0(bfm_index));
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi_tr_if_0(bfm_index));
```

AXI4 Example

```
set_config(AXI4_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi4_tr_if_0(bfm_index));
set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi4_tr_if_0(bfm_index));
```

get_config()

This nonblocking procedure gets the configuration of the slave BFM.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_config
(
  config_name   : in std_logic_vector(7 downto 0);
  config_val    : out std_logic_vector(**_MAX_BIT_SIZE-1 downto
  0)| integer;
  bfm_id        : in integer;
  path_id       : in *_path_t; --optional
  signal tr_if  : inout *_vhd_if_struct_t
);
```

Arguments

config_name (AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_MASTER_WRITE_DELAY
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS
- AXI_CONFIG_MAX_OUTSTANDING_WR
- AXI_CONFIG_MAX_OUTSTANDING_RD

(AXI4) Configuration name:
AXI4_CONFIG_SETUP_TIME
AXI4_CONFIG_HOLD_TIME
AXI4_CONFIG_BURST_TIMEOUT_FACTOR
AXI4_CONFIG_ENABLE_RLAST
AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
AXI4_CONFIG_ENABLE_ASSERTION
AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
AXI4_CONFIG_ENABLE_QOS
AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
AXI4_CONFIG_SLAVE_START_ADDR
AXI4_CONFIG_SLAVE_END_ADDR
AXI4_CONFIG_MAX_OUTSTANDING_WR
AXI4_CONFIG_MAX_OUTSTANDING_RD
AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE
AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE

config_val — Refer to "Slave BFM Configuration" on page 367 for description and valid values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns config_val

AXI3 Example

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
```

AXI4 Example

```
get_config(AXI4_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value,
     bfm_index, axi4_tr_if_0(bfm_index));
get_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
     axi4_tr_if_0(bfm_index));
```

create_slave_transaction()

This nonblocking procedure creates a slave transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns the *transaction_id* argument.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure create_slave_transaction
(
   transaction_id  : out integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221.

Protocol Transaction Fields

addr	Start address
size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;
lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;

cache	(AXI3) Burst cache: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW; (AXI4) Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;
prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
id	Burst ID.
burst_length	(Optional) Burst length. Default: 0.
data_words	Data words array.
write_strobes	Write strobes array: Each strobe 0 or 1.
resp	Response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;
region	(AXI4) Region identifier.

Operational Transaction Fields

qos	(AXI4) Quality-of-Service identifier.
addr_user	Address channel user data.
data_user	(AXI4) Data channel user data.
resp_user	(AXI4) Response channel user data.
read_or_write	Read or write transaction flag: **_TRANS_READ; **_TRANS_WRITE
gen_write_strobes	Correction of write strobes for invalid byte lanes: 0 = write_strobes passed through to protocol signals. 1 = write_strobes auto-corrected for invalid byte lanes (default).
operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING; (default)
delay_mode	Delay mode: AXI_VALID2READY; (default) AXI_TRANS2READY;
write_data_mode	Write data mode: **_DATA_AFTER_ADDRESS; The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS; (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.
address_valid_delay	Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction (default = 0).
data_valid_delay	Write data channel WVALID delay array measured in ACLK cycles for this transaction (default = 0 for all elements).
write_response_valid_delay	Write data channel BVALID delay measured in ACLK cycles for this transaction (default = 0).
address_ready_delay	Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction (default = 0).

	data_ready_delay	Read data channel RREADY delay measured in ACLK cycles for this transaction (default = 0).
	write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction (default = 0).
	data_beat_done	Data channel phase (beat) *done* array for this transaction.
	transaction_done	Transaction *done* flag for this transaction
Returns	transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221.

AXI3 Example

```
-- Create a slave transaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_3(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_3(bfm_index));
```

set_addr()

This nonblocking procedure sets the start address *addr* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_addr
(
   addr : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address of transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_addr()

This nonblocking procedure gets the start address *addr* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the create_slave_transaction() procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_addr
(
   addr : out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address of transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns addr

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_size()

This nonblocking procedure sets the burst *size* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_size
(
   size : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_size()

This nonblocking procedure gets the burst *size* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_size
(
   size : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

size	Burst size: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns size

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst size of the tr_id transaction.
get_size (size, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst size of the tr_id transaction.
get_size (size, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_burst()

This nonblocking procedure sets the *burst* type field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_burst
(
   burst: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst	Burst type: **_FIXED; **_INCR (default); **_WRAP; **_BURST_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_burst()

This nonblocking procedure gets the *burst* type field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_burst
(
   burst: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst	Burst type: **_FIXED; **_INCR; **_WRAP; **_BURST_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns burst

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst type of the tr_id transaction.
get_burst (burst, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst type of the tr_id transaction.
get_burst (burst, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_lock()

This nonblocking procedure sets the *lock* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_lock
(
   lock : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

lock	Burst lock: **_NORMAL (default); **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_lock()

This nonblocking procedure gets the *lock* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_lock
(
   lock : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

lock	Burst lock: **_NORMAL; **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

lock

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the lock field of the tr_id transaction.
get_lock(lock, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the lock field of the tr_id transaction.
get_lock(lock, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_cache()

This nonblocking procedure sets the *cache* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_cache
(
   cache: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

cache

(AXI3) Burst cache:
- AXI_NONCACHE_NONBUF; (default)
- AXI_BUF_ONLY;
- AXI_CACHE_NOALLOC;
- AXI_CACHE_BUF_NOALLOC;
- AXI_CACHE_RSVD0;
- AXI_CACHE_RSVD1;
- AXI_CACHE_WTHROUGH_ALLOC_R_ONLY;
- AXI_CACHE_WBACK_ALLOC_R_ONLY;
- AXI_CACHE_RSVD2;
- AXI_CACHE_RSVD3;
- AXI_CACHE_WTHROUGH_ALLOC_W_ONLY;
- AXI_CACHE_WBACK_ALLOC_W_ONLY;
- AXI_CACHE_RSVD4;
- AXI_CACHE_RSVD5;
- AXI_CACHE_WTHROUGH_ALLOC_RW;
- AXI_CACHE_WBACK_ALLOC_RW;

(AXI4) Burst cache:
- AXI4_NONMODIFIABLE_NONBUF; (default)
- AXI4_BUF_ONLY;
- AXI4_CACHE_NOALLOC;
- AXI4_CACHE_2;
- AXI4_CACHE_3;
- AXI4_CACHE_RSVD4;
- AXI4_CACHE_RSVD5;
- AXI4_CACHE_6;
- AXI4_CACHE_7;
- AXI4_CACHE_RSVD8;
- AXI4_CACHE_RSVD9;
- AXI4_CACHE_10;
- AXI4_CACHE_11;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_14;
- AXI4_CACHE_15;

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

Note

You do not normally use this procedure in a slave test program.

get_cache()

This nonblocking procedure gets the *cache* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_cache
(
   cache: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

cache

(AXI3) Burst cache:
- AXI_NONCACHE_NONBUF; (default)
- AXI_BUF_ONLY;
- AXI_CACHE_NOALLOC;
- AXI_CACHE_BUF_NOALLOC;
- AXI_CACHE_RSVD0;
- AXI_CACHE_RSVD1;
- AXI_CACHE_WTHROUGH_ALLOC_R_ONLY;
- AXI_CACHE_WBACK_ALLOC_R_ONLY;
- AXI_CACHE_RSVD2;
- AXI_CACHE_RSVD3;
- AXI_CACHE_WTHROUGH_ALLOC_W_ONLY;
- AXI_CACHE_WBACK_ALLOC_W_ONLY;
- AXI_CACHE_RSVD4;
- AXI_CACHE_RSVD5;
- AXI_CACHE_WTHROUGH_ALLOC_RW;
- AXI_CACHE_WBACK_ALLOC_RW;

(AXI4) Burst cache:
- AXI4_NONMODIFIABLE_NONBUF; (default)
- AXI4_BUF_ONLY;
- AXI4_CACHE_NOALLOC;
- AXI4_CACHE_2;
- AXI4_CACHE_3;
- AXI4_CACHE_RSVD4;
- AXI4_CACHE_RSVD5;
- AXI4_CACHE_6;
- AXI4_CACHE_7;
- AXI4_CACHE_RSVD8;
- AXI4_CACHE_RSVD9;
- AXI4_CACHE_10;
- AXI4_CACHE_11;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_14;
- AXI4_CACHE_15;

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

	path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
	tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
Returns	cache	

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the cache field of the tr_id transaction.
get_cache(cache, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the cache field of the tr_id transaction.
get_cache(cache, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_prot()

This nonblocking procedure sets the protection *prot* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_prot
(
   prot: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

prot	Protection: **_NORM_SEC_DATA (default); **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_prot()

This nonblocking procedure gets the protection *prot* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_prot
(
   prot: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

prot	Protection: **_NORM_SEC_DATA; **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

prot

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_id()

This nonblocking procedure sets the *id* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_id
(
  id: in integer;
  transaction_id  : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
  0) | integer;
  bfm_id : in integer;
  path_id : in *_path_t; --optional
  signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

id	Burst ID
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_id()

This nonblocking procedure gets the *id* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_id
(
 id: out integer;
 transaction_id  : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
 0) | integer;
 bfm_id : in integer;
 path_id : in *_path_t; --optional
 signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

id	Burst ID
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_burst_length()

This nonblocking procedures sets the *burst_length* field for a transaction uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Note

The *burst_length* field is the value that appears on the AWLEN and the ARLEN protocol signals. The number of data phases (beats) in a data burst is therefore *burst_length* + 1.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_burst_length
(
   burst_length : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0)
   | integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst_length	Burst length (default = 0).
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_burst_length()

This nonblocking procedures gets the *burst_length* field for a transaction uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Note

The *burst_length* field is the value that appears on the AWLEN and the ARLEN protocol signals. The number of data phases (beats) in a data burst is therefore *burst_length* + 1.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_burst_length
(
   burst_length : out std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0) | integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst_length	Burst length.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns burst_length

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst length field of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst length field of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_words()

This nonblocking procedure sets the read *data_words* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_data_words
(
   data_words: in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_words	Data words array.
index	(Optional) Array element number for *data_words*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the data_words field to 2 for the first read data phase (beat)
-- for the tr_id transaction.
set_data_words(2, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the data_words field to 3 for the second read data phase (beat)
-- for the tr_id transaction.
set_data_words(3, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the data_words field to 2 for the first read data phase (beat)
-- for the tr_id transaction.
set_data_words(2, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the data_words field to 3 for the second read data phase (beat)
-- for the tr_id transaction.
set_data_words(3, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_data_words()

This nonblocking procedure gets a *data_words* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_data_words
(
   data_words: out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_words	Data words array.
index	(Optional) Array element number for *data_words*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

data_words

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the data_words field of the first data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the data_words field of the second data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the data_words field of the first data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the data_words field of the second data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_write_strobes()

This nonblocking procedure sets the *write_strobes* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_write_strobes
(
   write_strobes : in std_logic_vector (**_MAX_BIT_SIZE-1 downto
   0) | integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_strobes	Write strobes array.
index	(Optional) Array element number for *write_strobes*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 123 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_write_strobes()

This nonblocking procedure gets the *write_strobes* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_write_strobes
(
   write_strobes : out std_logic_vector (**_MAX_BIT_SIZE-1 downto
   0) | integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_strobes	Write strobes array.
index	(Optional) Array element number for *write_strobes*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

write_strobes

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the write_strobes field of the first data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the write_strobes field of the second data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write_strobes field of the first data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the write_strobes field of the second data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_resp()

This nonblocking procedure sets the response *resp* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_resp
(
   resp: in std_logic_vector (**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

resp — Transaction response array:

**_OKAY = 0;
**_EXOKAY = 1;
**_SLVERR = 2;
**_DECERR = 3;

index — (Optional) Array element number for *resp*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the read response to AXI_OKAY for the first data phase (beat)
-- for the tr_id transaction.
set_resp(AXI_OKAY, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the read response to AXI_DECERR for the second data phase (beat)
-- for the tr_id transaction.
set_resp(AXI_DECERR, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the read response to AXI_OKAY for the first data phase (beat)
-- for the tr_id transaction.
set_resp(AXI4_OKAY, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the read response to AXI_DECERR for the second data phase (beat)
-- for the tr_id transaction.
set_resp(AXI4_DECERR, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_resp()

This nonblocking procedure gets a response *resp* field array element for a transaction uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4

get_resp
(
   resp: out std_logic_vector (**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

resp — Transaction response array:

**_OKAY = 0;
**_EXOKAY = 1;
**_SLVERR = 2;
**_DECERR = 3;

index — (Optional) Array element number for *resp*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

resp

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the response field of the first data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the response field of the second data phase (beat)
-- if the tr_id transaction.
get_resp(read_resp, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the response field of the first data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the response field of the second data phase (beat)
-- if the tr_id transaction.
get_resp(read_resp, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_addr_user()

This nonblocking procedures sets the user data *addr_user* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_addr_user
(
   addr_user : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr_user	User data in address phase.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_addr_user()

This nonblocking procedures gets the user data *addr_user* field for a transaction uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_addr_user
(
   addr_user : out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr_user	User data in address phase.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns addr_user

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel user data of the tr_id transaction.
get_addr_user(user_data, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel user data of the tr_id transaction.
get_addr_user(user_data, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_read_or_write()

This procedure sets the *read_or_write* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_read_or_write
(
   read_or_write: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

read_or_write	Read or write transaction: **_TRANS_READ = 0 **_TRANS_WRITE = 1
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_read_or_write()

This nonblocking procedure gets the *read_or_write* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_read_or_write
(
   read_or_write: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

read_or_write	Read or write transaction: **_TRANS_READ = 0 **_TRANS_WRITE = 1
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

read_or_write

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read_or_write field of tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read_or_write field of tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_gen_write_strobes()

This nonblocking procedure sets the *gen_write_strobes* field for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_gen_write_strobes
(
   gen_write_strobes: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

gen_write_strobes	Correction of write strobes for invalid byte lanes: 0 = *write_strobes* passed through to protocol signals. 1 = *write_strobes* auto-corrected for invalid byte lanes (default).
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_gen_write_strobes()

This nonblocking procedure gets the *gen_write_strobes* field for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_gen_write_strobes
(
   gen_write_strobes: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

gen_write_strobes	Correct write strobes flag: 0 = *write_strobes* passed through to protocol signals. 1 = *write_strobes* auto-corrected for invalid byte lanes.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns gen_write_strobes

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify the
transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify the
transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_operation_mode()

This nonblocking procedure sets the *operation_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_operation_mode
(
   operation_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING (default);
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the operation mode to nonblocking for the tr_id transaction.
set_operation_mode(AXI_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the operation mode to nonblocking for the tr_id transaction.
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

get_operation_mode()

This nonblocking procedure gets the *operation_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_operation_mode
(
   operation_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns operation_mode

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the operation mode of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the operation mode of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_delay_mode()

This AXI3 nonblocking procedure sets the *delay_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
set_delay_mode
(
   delay_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

delay_mode	Delay mode: AXI_VALID2READY (default); AXI_TRANS2READY;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the delay mode of the *VALID to *READY handshake for the
-- tr_id transaction.
set_delay_mode(AXI_VALID2READY, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

get_delay_mode()

This AXI3 nonblocking procedure gets the *delay_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
get_delay_mode
(
   delay_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

delay_mode	Delay mode: AXI_VALID2READY; AXI_TRANS2READY;
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns delay_mode

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the delay mode of the *VALID to *READY handshake of the
-- tr_id transaction
get_delay_mode(delay_mode, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

set_write_data_mode()

This nonblocking procedure sets the *write_data_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_write_data_mode
(
   write_data_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_data_mode — Write data mode:

**_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation.

**_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 221

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_write_data_mode()

This nonblocking procedure gets the *write_data_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_write_data_mode
(
   write_data_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_data_mode — Write data mode:

**_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation.

**_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

write_data_mode

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data mode of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data mode of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_address_valid_delay()

This nonblocking procedure sets the *address_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_address_valid_delay
(
   address_valid_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_valid_delay	Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_address_valid_delay()

This nonblocking procedure gets the *address_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_address_valid_delay
(
   address_valid_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_valid_delay	Address channel ARVALID/AWVALID delay in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns address_valid_delay

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_address_ready_delay()

This AXI3 nonblocking procedure sets the *address_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedures.

Prototype

```
set_address_ready_delay
(
   address_ready_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

address_ready_delay	Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Set the address channel ready delay to 3 ACLKs for the
-- tr_id transaction.
set_address_ready_delay(3, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

get_address_ready_delay()

This nonblocking procedure gets the *address_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_address_ready_delay
(
   address_ready_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_ready_delay	Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns address_ready_delay

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_valid_delay()

This nonblocking procedure sets the *data_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_data_valid_delay
(
   data_valid_delay: in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_valid_delay	Read data channel array to hold RVALID delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element number for *data_valid_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_write_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the read channel RVALID delay to 3 ACLK cycles for the first data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(3, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the read channel RVALID delay to 2 ACLK cycles for the second data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(2, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_write_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the read channel RVALID delay to 3 ACLK cycles for the first data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(3, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the read channel RVALID delay to 2 ACLK cycles for the second data
-- phase (beat) of the tr_id transaction.
set_data_valid_delay(2, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_data_valid_delay()

This nonblocking procedure sets the *data_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_data_valid_delay
(
   data_valid_delay: out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_valid_delay	Data channel array to hold RVALID/WVALID delays measured in ACLK cycles for this transaction.
index	(Optional) Array element number for *data_valid_delay*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

data_valid_delay

AXI3 Example

```
-- Create a slave transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....
-- Get the write channel WVALID delay for the first data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the write channel WVALID delay for the second data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....
-- Get the write channel WVALID delay for the first data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the write channel WVALID delay for the second data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_ready_delay()

This AXI3 nonblocking procedure sets the *data_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by *create_slave_transaction()*.

Prototype

```
set_data_ready_delay
(
   data_ready_delay: in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

data_ready_delay	Write data channel array to hold WREADY delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element number for *data_ready_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
-- Set the write data channel WREADY delay to 3 ACLK cycles for the first
-- dataphase (beat) of the tr_id transaction.
set_data_ready_delay(3, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));
-- Set the write data channel WREADY delay to 2 ACLK cycles for the second
-- data phase (beat) of the tr_id transaction.
set_data_ready_delay(2, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

get_data_ready_delay()

This nonblocking procedure gets the *data_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_data_ready_delay
(
   data_ready_delay: out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_ready_delay	Data channel array to hold RREADY/WREADY delay measured in ACLK cycles for this transaction.
index	(Optional) Array element number for *data_ready_delay*
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the first
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the second
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the first
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the second
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_write_response_valid_delay()

This nonblocking procedure sets the *write_response_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
set_write_response_valid_delay
(
   write_response_valid_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_valid_delay	Write data channel BVALID delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the write response channel BVALID delay to 3 ACLK cycles for the
-- tr_id transaction.
set_write_response_valid_delay(3, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the write response channel BVALID delay to 3 ACLK cycles for the
-- tr_id transaction.
set_write_response_valid_delay(3, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

get_write_response_valid_delay()

This nonblocking procedure gets the *write_response_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_write_response_valid_delay
(
   write_response_valid_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_valid_delay	Write data channel BVALID delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

write_response_valid_delay

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id,
bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id,
bfm_index, axi4_tr_if_0(bfm_index));
```

set_write_response_ready_delay()

This AXI3 nonblocking procedure sets the *write_response_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
set_write_response_ready_delay
(
   write_response_ready_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a slave test program.

get_write_response_ready_delay()

This nonblocking procedure gets the *write_response_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_write_response_ready_delay
(
   write_response_ready_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

write_response_ready_delay

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_beat_done()

This nonblocking procedure sets the *data_beat_done* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_data_beat_done
(
   data_beat_done : in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_beat_done	Write data channel phase (beat) *done* array for this transaction.
index	(Optional) Array element number for *data_beat_done*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Set the write data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Set the write data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Set the write data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Set the write data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
set_data_beat_done(1, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_data_beat_done()

This nonblocking procedure gets the *data_beat_done* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_data_beat_done
(
   data_beat_done : out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_beat_done	Data channel phase (beat) *done* array for this transaction
index	(Optional) Array element number for *data_beat_done*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

data_beat_done

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_transaction_done()

This nonblocking procedure sets the *transaction_done* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_transaction_done
(
   transaction_done : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Set the read transaction_done flag of the tr_id transaction.
set_transaction_done(1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction.
-- Creation returns tr_id to identify the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Set the slave transaction_done flag of the tr_id transaction.
set_transaction_done(1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_transaction_done()

This nonblocking procedure gets the *transaction_done* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_transaction_done
(
   transaction_done : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns transaction_done

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

execute_read_data_burst()

This procedure executes a read data burst that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure. This burst can be blocking (default), or nonblocking, defined by the transaction record *operation_mode* field.

It calls the *execute_read_data_phase()* procedure for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure execute_read_data_burst
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Execute the read data burst for the tr_id transaction.
execute_read_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Execute the read data burst for the tr_id transaction.
execute_read_data_burst(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

execute_read_data_phase()

This procedure executes a read data phase that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

The *execute_read_data_phase()* sets the RVALID protocol signal at the appropriate time defined by the transaction record *data_valid_delay* field and sets the *data_beat_done* array *index* element field to 1 when the phase completes. If this is the last data phase (beat) of the burst, it also sets the *transaction_done* field on completion.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure execute_read_data_phase
(
   transaction_id  : in integer;
   index : in integer; --optional
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
index	(Optional) Data phase (beat) number.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....
-- Execute the read data phase for the first beat of the
-- tr_id transaction.
execute_read_data_phase(tr_id, 0, bfm_index, axi_tr_if_0(bfm_index));

-- Execute the read data phase for the second beat of the
-- tr_id transaction.
execute_read_data_phase(tr_id, 1, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....
-- Execute the read data phase for the first beat of the
-- tr_id transaction.
execute_read_data_phase(tr_id, 0, bfm_index, axi4_tr_if_0(bfm_index));

-- Execute the read data phase for the second beat of the
-- tr_id transaction.
execute_read_data_phase(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```

execute_write_response_phase()

This procedure executes a write response phase that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure. This phase can be blocking (default) or nonblocking, defined by the transaction record *operation_mode* field.

It sets the BVALID protocol signal at the appropriate time defined by the transaction record *write_response_valid_delay* field, and sets the *data_beat_done* array *index* element field on completion. If this is the last data phase (beat) of the burst, it also sets the *transaction_done* field on completion.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure execute_write_response_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_2(bfm_index));

....

-- Execute the write response phase of the tr_id transaction.
execute_write_response_phase(tr_id, bfm_index, axi_tr_if_2(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_2(bfm_index));

....

-- Execute the write response phase of the tr_id transaction.
execute_write_response_phase(tr_id, bfm_index, axi4_tr_if_2(bfm_index));
```

get_write_addr_phase()

Note

This blocking procedure gets a write address phase uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure. For AXI3, the *get_write_addr_phase()* also sets the AWREADY protocol signal at the appropriate time defined by the transaction record *address_ready_delay* field.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_write_addr_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; -- Optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write address phase of the tr_id transaction.
get_write_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write address phase of the tr_id transaction.
get_write_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_addr_phase()

This blocking procedure gets a read address phase uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure.

Note

For AXI3, the *get_read_addr_phase()* also sets the ARREADY protocol signal at the appropriate time defined by the transaction record *address_ready_delay* field.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure get_read_addr_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; -- Optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read address phase of the tr_id transaction.
get_read_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read address phase of the tr_id transaction.
get_read_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_data_phase()

This blocking procedure gets a write data phase that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure. It sets the *data_beat_done* array *index* element to 1 when the phase completes. If this is the last data phase of the burst, then it returns the *last* argument set to 1.

Note

For AXI3, the *get_write_data_phase()* also sets the WREADY protocol signal at the appropriate time defined by the transaction record *data_ready_delay* array *index* element when the phase completes.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_write_data_phase
(
   transaction_id  : in integer;
   index : in integer; --optional
   last : out integer;
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
index	(Optional) Data phase (beat) number.
last	Last data phase (beat) of the burst: 0 = data burst not complete 1 = data burst complete
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

last

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data phase for the first beat of the tr_id transaction.
get_write_data_phase(tr_id, 0, last, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data phase for the second beat of the tr_id transaction.
get_write_data_phase(tr_id, 1, last, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data phase for the first beat of the tr_id transaction.
get_write_data_phase(tr_id, 0, last, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data phase for the second beat of the tr_id transaction.
get_write_data_phase(tr_id, 1, last, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_data_burst()

This blocking procedure gets a write data burst that is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure.

It calls the *get_write_data_phase()* procedure for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_write_data_burst
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data burst for the tr_id transaction.
get_write_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data burst for the tr_id transaction.
get_write_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

get_read_addr_cycle()

This blocking AXI4 procedure waits until the read address channel ARVALID signal is asserted.

Prototype

```
procedure get_read_addr_cycle
(
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

AXI3 BFM

Note

The *get_read_addr_cycle()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Wait for the ARVALID signal to be asserted.
bfm.get_read_addr_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```

execute_read_addr_ready()

This AXI4 procedure executes a read address ready by placing the *ready* argument value onto the ARREADY signal. It will block (default) for one ACLK period.

Prototype

```
procedure execute_read_addr_ready
(
   ready : in integer;
   bfm_id          : in integer;
   path_id         : in axi4_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
non_blocking_mode	(Optional) Nonblocking mode: 0 = Nonblocking 1 = Blocking (default)
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_0 AXI4_PATH_1 AXI4_PATH_2 AXI4_PATH_3 AXI4_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 BFM

Note

The *execute_read_addr_ready()* task is not available in the AXI3 BFM. Use the *get_read_addr_phase()* task along with the transaction record *address_ready_delay* field.

AXI4 Example

```
-- Set the ARREADY signal to 1 and block for 1 ACLK cycle
execute_read_addr_ready(1, 1, index, AXI4_PATH_6, axi4_tr_if_6(index));
```

get_read_data_ready()

This blocking AXI4 procedure returns the value of the read data channel RREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_read_data_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the RREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_read_data_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the RREADY signal value
bfm.get_read_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_addr_cycle()

This blocking AXI4 procedure waits until the write address channel AWVALID signal is asserted.

Prototype

```
procedure get_write_addr_cycle
(
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 BFM

Note

The *get_write_addr_cycle()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Wait for the AWVALID signal to be asserted.
bfm.get_write_addr_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```

execute_write_addr_ready()

This AXI4 procedure executes a write address ready by placing the *ready* argument value onto the AWREADY signal. It will block for one ACLK period.

Prototype

```
procedure execute_write_addr_ready
(
   ready : in integer;
   bfm_id          : in integer;
   path_id         : in axi4_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
index	(Optional) Data phase (beat) number.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_0 AXI4_PATH_1 AXI4_PATH_2 AXI4_PATH_3 AXI4_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

AXI3 BFM

Note

The *execute_write_addr_ready()* task is not available in the AXI3 BFM. Use the *get_write_addr_phase()* task along with the transaction record *address_ready_delay* field.

AXI4 Example

```
-- Set the AWREADY signal to 1 and block for 1 ACLK cycle
execute_write_addr_ready(1, 1, index, AXI4_PATH_5, axi4_tr_if_5(index));
```

get_write_data_cycle()

This blocking AXI4 procedure waits until the write data channel WVALID signal is asserted.

Prototype

```
procedure get_write_data_cycle
(
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 BFM

Note

The *get_write_data_cycle()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Wait for the WVALID signal to be asserted.
bfm.get_write_data_cycle(bfm_index, axi4_tr_if_0(bfm_index));
```

execute_write_data_ready()

This AXI4 procedure executes a write data ready by placing the *ready* argument value onto the WREADY signal. It blocks for one ACLK period.

Prototype

```
procedure execute_write_data_ready
(
   ready : in integer;
   bfm_id          : in integer;
   path_id         : in axi4_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
non_blocking_mode	(Optional) Nonblocking mode: 0 = Nonblocking 1 = Blocking (default)
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_0 AXI4_PATH_1 AXI4_PATH_2 AXI4_PATH_3 AXI4_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 BFM

Note

The *execute_write_data_ready()* task is not available in the AXI3 BFM. Use the *get_write_data_phase()* task along with the transaction record *address_ready_delay* field.

AXI4 Example

```
-- Set the WREADY signal to 1 and block for 1 ACLK cycle
execute_write_data_ready(1, 1, index, AXI4_PATH_7, axi4_tr_if_7(index));
```

get_write_resp_ready()

This blocking AXI4 procedure returns the value of the write response channel BREADY signal using the *ready* argument. It blocks for one ACLK period.

Prototype

```
procedure get_write_resp_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the RREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_write_resp_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the BREADY signal value
bfm.get_write_resp_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

push_transaction_id()

This nonblocking procedure pushes a transaction ID into the back of a queue. The transaction is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure. The queue is identified by the *queue_id* argument.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure push_transaction_id
(
   transaction_id  : in integer;
   queue_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

queue_id — Queue identifier:

**_QUEUE_ID_0
**_QUEUE_ID_1
**_QUEUE_ID_2
**_QUEUE_ID_3
**_QUEUE_ID_4
AXI4_QUEUE_ID_5
AXI4_QUEUE_ID_6
AXI4_QUEUE_ID_7

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI_QUEUE_ID_1, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
```

pop_transaction_id()

This nonblocking (unless queue is empty) procedure pops a transaction ID from the front of a queue. The transaction is uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure. The queue is identified by the *queue_id* argument.

If the queue is empty, then it will block until an entry becomes available.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure pop_transaction_id
(
   transaction_id  : in integer;
   queue_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

queue_id — Queue identifier:

- **_QUEUE_ID_0
- **_QUEUE_ID_1
- **_QUEUE_ID_2
- **_QUEUE_ID_3
- **_QUEUE_ID_4
- AXI4_QUEUE_ID_5
- AXI4_QUEUE_ID_6
- AXI4_QUEUE_ID_7

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

- **_PATH_0
- **_PATH_1
- **_PATH_2
- **_PATH_3
- **_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI_QUEUE_ID_1, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
```

print()

This nonblocking procedure prints a transaction record that is uniquely identified by the *transaction_id* argument previously created by the create_slave_transaction() procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure print
(
   transaction_id  : in integer;
   print_delays : in integer; --optional
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

print_delays — (Optional) Print delay values flag:

0 = do not print the delay values (default).
1 = print the delay values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```

destruct_transaction()

This blocking procedure removes a transaction record for cleanup purposes and memory management, uniquely identified by the *transaction_id* argument previously created by the *create_slave_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure destruct_transaction
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a slave transaction. Creation returns tr_id to identify
-- the transaction.
create_slave_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

wait_on()

This blocking procedure waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count*.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure wait_on
(
   phase           : in integer;
   count: in integer; --optional
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

phase	Wait for: **_CLOCK_POSEDGE **_CLOCK_NEGEDGE **_CLOCK_ANYEDGE **_CLOCK_0_TO_1 **_CLOCK_1_TO_0 **_RESET_POSEDGE **_RESET_NEGEDGE **_RESET_ANYEDGE **_RESET_0_TO_1 **_RESET_1_TO_0
count	(Optional) Wait for a number of events to occur set by *count*. (default = 1)
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
wait_on(AXI_RESET_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
wait_on(AXI_CLOCK_POSEDGE, 10, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
wait_on(AXI4_RESET_POSEDGE, bfm_index, axi4_tr_if_0(bfm_index));
wait_on(AXI4_CLOCK_POSEDGE, 10, bfm_index, axi4_tr_if_0(bfm_index));
```

Helper Functions

AMBA AXI protocols typically provide a start address only in a transaction, with the following addresses for each byte of a data burst calculated using the size, length, and type transaction fields of the transaction. Helper functions provide you with a simple interface to set and get actual address/data values.

get_write_addr_data()

This nonblocking procedure returns the actual address *addr* and *data* of a particular byte in a write data burst . It also returns the maximum number of bytes (*dynamic_size*) in the write data phase (beat). It is used in a slave test program as a helper procedure to store a byte of data at a particular address in the slave memory. If the corresponding *index* does not exist, then this function returns *false*; otherwise, it returns *true*.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure get_write_addr_data
(
   transaction_id  : in integer;
   index           : in integer;
   byte_index      : in integer;
   dynamic_size    : out integer;
   addr            : out std_logic_vector(**_MAX_BIT_SIZE-1
   downto 0);
   data            : out std_logic_vector(7 downto 0);
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
index	Data words array element number.
byte_index	Data byte number in a data phase (beat)
dynamic_size	Number of data bytes in a data phase (beat).
addr	Data byte address.
data	Write data byte.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

	tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
Returns	dynamic_size	
	addr	
	data	

AXI3 Example

```
-- Wait for the write data burst to complete for the tr_id transaction.
get_write_data_burst(tr_id, bfm_index, axi_tr_if_1(bfm_index));

-- Get the burst length of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index, axi_tr_if_1(bfm_index));

-- Loop for the length of the data burst.
for i in  0 to burst_length loop

   -- Get the address, first data byte and byte length for the
   -- data phase (beat)
   get_write_addr_data(tr_id, i, 0, byte_length, addr, data, bfm_index,
                       axi_tr_if_1(bfm_index));

   -- Store the first data byte in the slave memory using the
   -- slave test program do_byte_write procedure
   do_byte_write(addr, data);

   -- Get the remaining bytes of the write data phase (beat)
   -- and store thenm in the slave memory.
   if byte_length > 1 then
      for j in 1 to byte_length-1 loop

         get_write_addr_data(write_trans, i, j, byte_length, addr, data,
                             index, axi_tr_if_1(index));
         do_byte_write(addr, data);
      end loop;
   end if;
end loop;
```

AXI4 Example

```
-- Wait for the write data burst to complete for the tr_id transaction.
get_write_data_burst(tr_id, bfm_index, axi4_tr_if_1(bfm_index));

-- Get the burst length of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index,
      axi4_tr_if_1(bfm_index));

-- Loop for the length of the data burst.
for i in  0 to burst_length loop

   -- Get the address, first data byte and byte length for the
   -- data phase (beat)
   get_write_addr_data(tr_id, i, 0, byte_length, addr, data, bfm_index,
                       axi4_tr_if_1(bfm_index));

   -- Store the first data byte in the slave memory using the
   -- slave test program do_byte_write procedure
   do_byte_write(addr, data);

   -- Loop for the number of bytes in the write data phase (beat)
   -- given by the byte_length
   if byte_length > 1 then
      for j in 1 to byte_length-1 loop

         -- Get the remaining bytes of the write data phase (beat)
         -- and store thenm into the slave memory.
         get_write_addr_data(write_trans, i, j, byte_length, addr, data,
                             index, axi4_tr_if_1(index));
         do_byte_write(addr, data);
      end loop;
   end if;
end loop;
```

get_read_addr()

This nonblocking procedure returns the actual address *addr* a particular byte in a read data transaction. It also returns the maximum number of bytes (*dynamic_size*) in the read data phase (beat). It is used in a slave test program as a helper procedure to return the address of a data byte in the slave memory.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_read_addr
(
   transaction_id  : in integer;
   index           : in integer;
   byte_index      : in integer;
   dynamic_size    : out integer;
   addr           : out std_logic_vector(**_MAX_BIT_SIZE-1
   downto 0);
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
index	Data words array element number.
byte_index	Data byte number in a data phase (beat)
dynamic_size	Number of data bytes in a data phase (beat).
addr	Data byte address.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

dynamic_size

addr

AXI3 Example

```
-- Wait for the write data burst to complete for the transaction.
get_burst_length(burst_length, read_trans, index, AXI_PATH_4,
                     axi_tr_if_4(index));

-- Loop for the length of the data burst.
for i in  0 to burst_length loop

   -- Get the byte address and number of bytes in the data phase (beat).
   get_read_addr(read_trans,  i, 0, byte_length, addr, index, AXI_PATH_4,
                  axi_tr_if_4(index));

   -- Retrieve the first data byte from the slave memory using the
   -- slave test program do_byte_read procedure.
   do_byte_read(addr, data);

   -- Set the first read data byte for the read_trans transaction.
   set_read_data(read_trans, i, 0, byte_length, addr, data, index,
                    AXI_PATH_4, axi_tr_if_4(index));

   -- Loop for the number of bytes in the data phase (beat)
   -- given by the byte_length.
   if byte_length > 1 then
      for j in 1 to byte_length-1 loop

         -- Get the next read data byte address.
         get_read_addr(read_trans,  i, j, byte_length, addr, index,
                          AXI_PATH_4, axi_tr_if_4(index));

         -- Retrieve the next data byte from the slave memory using the
         -- slave test program do_byte_read procedure.
         do_byte_read(addr, data);

         -- Set the next read data byte for the read_trans transaction.
         set_read_data(read_trans, i, j, byte_length, addr, data, index,
                        AXI_PATH_4, axi_tr_if_4(index));

      end loop;
   end if;
end loop;
```

AXI4 Example

```
-- Get the burst length of the read_trans transaction.
get_burst_length(burst_length, read_trans, index, AXI4_PATH_4,
                     axi4_tr_if_4(index));

-- Loop for the length of the data burst.
for i in  0 to burst_length loop

   -- Get the byte address and number of bytes in the data phase (beat).
   get_read_addr(read_trans,  i, 0, byte_length, addr, index,
                        AXI4_PATH_4, axi4_tr_if_4(index));

   -- Retrieve the first data byte from the slave memory using the
   -- slave test program do_byte_read procedure.
   do_byte_read(addr, data);

   -- Set the first read data byte for the read_trans transaction_id.
   set_read_data(read_trans, i, 0, byte_length, addr, data, index,
                     AXI4_PATH_4, axi4_tr_if_4(index));

   -- Loop for the number of bytes in the data phase (beat)
   -- given by the byte_length.
   if byte_length > 1 then
      for j in 1 to byte_length-1 loop

         -- Get the next read data byte address.
         get_read_addr(read_trans,  i, j, byte_length, addr, index,
                           AXI4_PATH_4, axi4_tr_if_4(index));

         -- Retrieve the next data byte from the slave memory using the
         -- slave test program do_byte_read procedure
         do_byte_read(addr, data);

         -- Set the read data byte for the read_trans transaction.
         set_read_data(read_trans, i, j, byte_length, addr, data,
                              index, AXI4_PATH_4, axi4_tr_if_4(index));
      end loop;
   end if;
end loop;
```

set_read_data()

This nonblocking procedure sets a read *data* byte in a read transaction prior to execution. It is used in a slave test program as a helper procedure to set the read data retrieved from the slave memory into the relevant byte of a read data phase.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure set_read_data
(
   transaction_id  : in integer;
   index           : in integer;
   byte_index      : in integer;
   dynamic_size    : in integer;
   addr            : in std_logic_vector(**_MAX_BIT_SIZE-1
   downto 0);
   data            : in std_logic_vector(7 downto 0);
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
index	(Optional) Data byte array element number.
byte_index	Data byte index number of a particular data phase (beat).
dynamic_size	Maximum number of bytes in a particular data phase (beat).
addr	Read address.
data	Read data byte.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
-- Wait for the write data burst to complete for the transaction.
get_burst_length(burst_length, read_trans, index, AXI_PATH_4,
                     axi_tr_if_4(index));

-- Loop for the length of the data burst.
for i in  0 to burst_length loop

   -- Get the byte address and number of bytes in the data phase (beat).
   get_read_addr(read_trans,  i, 0, byte_length, addr, index, AXI_PATH_4,
                  axi_tr_if_4(index));

   -- Retrieve the first data byte from the slave memory using the
   -- slave test program do_byte_read procedure.
   do_byte_read(addr, data);

   -- Set the first read data byte for the read_trans transaction.
   set_read_data(read_trans, i, 0, byte_length, addr, data, index,
                     AXI_PATH_4, axi_tr_if_4(index));

   -- Loop for the number of bytes in the data phase (beat)
   -- given by the byte_length.
   if byte_length > 1 then
      for j in 1 to byte_length-1 loop

         -- Get the next read data byte address.
         get_read_addr(read_trans,  i, j, byte_length, addr, index,
                           AXI_PATH_4, axi_tr_if_4(index));

         -- Retrieve the next data byte from the slave memory using the
         -- slave test program do_byte_read procedure.
         do_byte_read(addr, data);

         -- Set the next read data byte for the read_trans transaction.
         set_read_data(read_trans, i, j, byte_length, addr, data, index,
                        AXI_PATH_4, axi_tr_if_4(index));

      end loop;
   end if;
end loop;
```

AXI4 Example

```
-- Get the burst length of the read_trans transaction.
get_burst_length(burst_length, read_trans, index, AXI4_PATH_4,
                     axi4_tr_if_4(index));

-- Loop for the length of the data burst.
for i in  0 to burst_length loop

   -- Get the byte address and number of bytes in the data phase (beat).
   get_read_addr(read_trans,  i, 0, byte_length, addr, index,
                        AXI4_PATH_4, axi4_tr_if_4(index));

   -- Retrieve the first data byte from the slave memory using the
   -- slave test program do_byte_read procedure.
   do_byte_read(addr, data);

   -- Set the first read data byte for the read_trans transaction_id.
   set_read_data(read_trans, i, 0, byte_length, addr, data, index,
                     AXI4_PATH_4, axi4_tr_if_4(index));

   -- Loop for the number of bytes in the data phase (beat)
   -- given by the byte_length.
   if byte_length > 1 then
      for j in 1 to byte_length-1 loop

         -- Get the next read data byte address.
         get_read_addr(read_trans,  i, j, byte_length, addr, index,
                           AXI4_PATH_4, axi4_tr_if_4(index));

         -- Retrieve the next data byte from the slave memory using the
         -- slave test program do_byte_read procedure
         do_byte_read(addr, data);

         -- Set the read data byte for the read_trans transaction.
         set_read_data(read_trans, i, j, byte_length, addr, data,
                              index, AXI4_PATH_4, axi4_tr_if_4(index));
      end loop;
   end if;
end loop;
```

Chapter 10
VHDL AXI3 and AXI4 Monitor BFMs

This chapter provides information about the VHDL AXI3 and AXI4 monitor BFMs. Each BFM has an API containing procedures that configure the BFM and access the dynamic Transaction Record during the lifetime of a transaction.

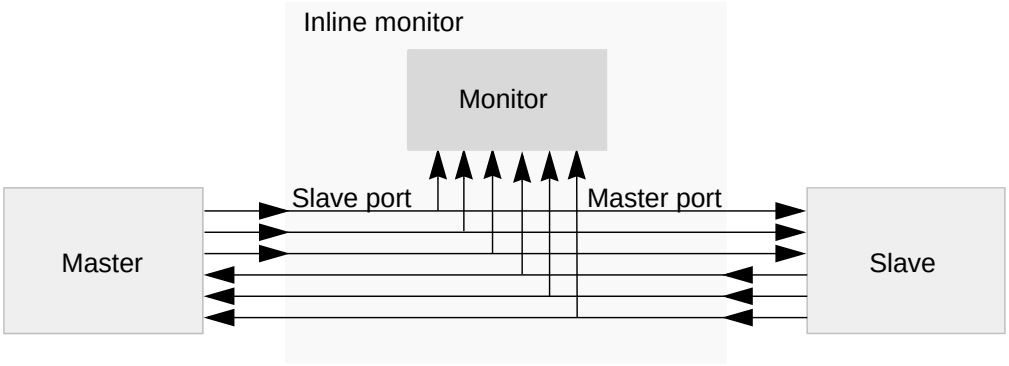
Note

Due to AXI3 protocol specification changes, for some BFM tasks, you reference the AXI3 BFM by specifying AXI instead of AXI3.

Inline Monitor Connection

The connection of a monitor BFM to a test environment differs from that of a master and slave BFM. It is wrapped in an inline monitor interface and connected inline, between a master and slave, as shown in Figure 10-1. It has separate master and slave ports and monitors protocol traffic between a master and slave. By construction, the monitor has access to all the facilities provided by the monitor BFM.

Figure 10-1. Inline Monitor Connection Diagram



Monitor BFM Protocol Support

The AXI3 monitor BFM supports the AMBA AXI3 protocol with restrictions detailed in "Protocol Restrictions" on page 19. In addition to the standard protocol, it supports user sideband signals AWUSER and ARUSER.

The AXI4 monitor BFM supports the AMBA AXI4 protocol with restrictions detailed in "Protocol Restrictions" on page 19.

Monitor Timing and Events

For detailed timing diagrams of the protocol bus activity and details of the following monitor BFM API timing and events, refer to the relevant AMBA AXI Protocol Specification chapter,

The AMBA AXI Protocol Specification does not define any timescale or clock period with signal events sampled and driven at rising ACLK edges. Therefore, the monitor BFM does not contain any timescale, timeunit, or timeprecision declarations with the signal setup and hold times specified in units of simulator time-steps.

Monitor BFM Configuration

The monitor BFM supports the full range of signals defined for the AMBA AXI Protocol Specification. The BFM has parameters you can use to configure the widths of the address, ID and data signals, and transaction fields to configure timeout factors, slave exclusive support, and setup and hold times, and so on.

You can change the address, ID and data signal widths from their default settings by assigning them new values, usually performed in the top-level module of the test bench. These new values are then passed into the monitor BFM via a parameter port list of the monitor BFM component.

Table 10-1 lists the parameter names for the address, ID and data, and their default values..

Table 10-1. Signal Parameters

Signal Width Parameter	Description
**_ADDRESS_WIDTH	Address signal width in bits. This applies to the ARADDR and AWADDR signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 32.
**_RDATA_WIDTH	Read data signal width in bits. This applies to the RDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_WDATA_WIDTH	Write data signal width in bits. This applies to the WDATA signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 64.
**_ID_WIDTH	ID signal width in bits. This applies to the RID and WID signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 4.
AXI4_USER_WIDTH	(AXI4) User data signal width in bits. This applies to the ARUSER, AWUSER, RUSER, WUSER, and BUSER signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 8.

Table 10-1. Signal Parameters (cont.)

Signal Width Parameter	Description
AXI4_REGION_MAP_SIZE	(AXI4) Region signal width in bits. This applies to the ARREGION and AWREGION signals. Refer to the AMBA AXI Protocol Specification for more details. Default: 16.
index	Uniquely identifies a monitor BFM instance. It must be set to a different value for each monitor BFM in the system. Default: 0.
READ_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding read transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
WRITE_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding write transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
COMBINED_ACCEPTANCE_ CAPABILITY	The maximum number of outstanding combined read and write transactions that can be accepted by the monitor BFM. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. Default: 16.
USE_*	(AXI4) Each protocol signal connection to the monitor BFM can be enabled or disabled. This parameter is set with the Qsys Parameter Editor. See “Running the Qsys Tool” on page 676. for details. 0 = disabled. 1 = enabled (default).

A monitor BFM has configuration fields that you can set via the *set_config()* function to configure timeout factors, slave exclusive support, setup and hold times, and so on. You can also get the value of a configuration field via the *get_config()* function. The full list of configuration fields is described in Table 10-2.

Table 10-2. Monitor BFM Configuration

Configuration Field	Description
Timing Variables	
**_CONFIG_SETUP_TIME	The setup-time prior to the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.

Table 10-2. Monitor BFM Configuration (cont.)

Configuration Field	Description
**_CONFIG_HOLD_TIME	The hold-time after the active edge of ACLK, in units of simulator time-steps for all signals.[1] Default: 0.
**_CONFIG_MAX_TRANSACTION_ TIME_FACTOR	The maximum timeout duration for a read/write transaction in clock cycles. Default: 100000.
AXI_CONFIG_TIMEOUT_MAX_DATA_ TRANSFER	(AXI3) The maximum number of write data beats that the AXI3 BFM can generate as part of write data burst of write transfer. Default: 1024.
_CONFIG_BURST_TIMEOUT_FACTOR	The maximum delay between the individual phases of a read/write transaction in clock cycles. Default: 10000.
**_CONFIG_MAX_LATENCY_AWVALID_ ASSERTION_TO_AWREADY	The maximum timeout duration from the assertion of AWVALID to the assertion of AWREADY in clock. periods. Default: 10000.
**_CONFIG_MAX_LATENCY_ARVALID_ ASSERTION_TO_ARREADY	The maximum timeout duration from the assertion of ARVALID to the assertion of ARREADY in clock periods. Default:10000.
**_CONFIG_MAX_LATENCY_RVALID_ ASSERTION_TO_RREADY	The maximum timeout duration from the assertion of RVALID to the assertion of RREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_BVALID_ ASSERTION_TO_BREADY	The maximum timeout duration from the assertion of BVALID to the assertion of BREADY in clock periods. Default: 10000.
**_CONFIG_MAX_LATENCY_WVALID_ ASSERTION_TO_WREADY	The maximum timeout duration from the assertion of WVALID to the assertion of WREADY in clock periods. Default: 10000.
Master Attributes	
AXI4_CONFIG_ENABLE_RLAST	(AXI4) Configures the support for the optional RLAST signal. 0 = disabled 1 = enabled (default)
Slave Attributes	

Table 10-2. Monitor BFM Configuration (cont.)

Configuration Field	Description
**_CONFIG_SLAVE_START_ADDR	Configures the start address map for the slave.
**_CONFIG_SLAVE_END_ADDR	Configures the end address map for the slave.
**_CONFIG_READ_DATA_REORDERING_DEPTH	The slave read reordering depth. Refer to the AMBA AXI Protocol Specification for more details. Default: 1.
Error Detection	
**_CONFIG_ENABLE_ALL_ASSERTIONS	Global enable/disable of all assertion checks in the BFM. 0 = disabled 1 = enabled (default)
**_CONFIG_ENABLE_ASSERTION	Individual enable/disable of assertion check in the BFM. 0 = disabled 1 = enabled (default)

[1.] Refer to Monitor Timing and Events for details of simulator time-steps.

Monitor Assertions

The monitor BFM performs protocol error checking via built-in assertions.

Note
The built-in BFM assertions are independent of programming language and simulator.

AXI3 Assertion Configuration

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI_CONFIG_ENABLE_ALL_ASSERTIONS,0,bfm_index,
axi_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI_MAX_BIT_SIZE-1
downto 0);
```

```
-- Get the current value of the assertion bit vector
get_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi_tr_if_0(bfm_index));

-- Assign the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI_LOCK_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI_CONFIG_ENABLE_ASSERTION bit vector with the AXI_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI_LOCK_CHANGED_BEFORE_AWREADY assertion, following the above code sequence, assign the assertion in the AXI_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to "AXI4 Assertions" on page 738.

AXI4 Assertion Configuration

By default, all built-in assertions are enabled in the monitor BFM. To globally disable them in the monitor BFM, use the *set_config()* command as the following example illustrates:

```
set_config(AXI4_CONFIG_ENABLE_ALL_ASSERTIONS,0,bfm_index,
axi4_tr_if_0(bfm_index));
```

Alternatively, you can disable individual built-in assertions by using a sequence of *get_config()* and *set_config()* commands on the respective assertion. For example, to disable assertion checking for the AWLOCK signal changing between the AWVALID and AWREADY handshake signals, use the following sequence of commands:

```
-- Define a local bit vector to hold the value of the assertion bit vector
variable config_assert_bitvector : std_logic_vector(AXI4_MAX_BIT_SIZE-1
downto 0);

-- Get the current value of the assertion bit vector
get_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4_tr_if_0(bfm_index));

-- Assign the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion bit to 0
config_assert_bitvector(AXI4_LOCK_CHANGED_BEFORE_AWREADY) := '0';

-- Set the new value of the assertion bit vector
set_config(AXI4_CONFIG_ENABLE_ASSERTION, config_assert_bitvector,
bfm_index, axi4_tr_if_0(bfm_index));
```

Note

Do not confuse the AXI4_CONFIG_ENABLE_ASSERTION bit vector with the AXI4_CONFIG_ENABLE_ALL_ASSERTIONS global enable/disable.

To re-enable the AXI4_LOCK_CHANGED_BEFORE_AWREADY assertion, following the above code sequence, assign the assertion in the AXI4_CONFIG_ENABLE_ASSERTION bit vector to 1.

For a complete listing of assertions, refer to "AXI4 Assertions" on page 738.

VHDL Monitor API

This section describes the VHDL Monitor API.

set_config()

This nonblocking procedure sets the configuration of the monitor BFM.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure set_config
(
   config_name    : in std_logic_vector(7 downto 0);
   config_val     : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0)|integer;
   bfm_id         : in integer;
   path_id        : in *_path_t; --optional
   signal tr_if   : inout *_vhd_if_struct_t
);
```

Arguments

config_name	(AXI3) Configuration name: AXI_CONFIG_SETUP_TIME AXI_CONFIG_HOLD_TIME AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER AXI_CONFIG_BURST_TIMEOUT_FACTOR AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME AXI_CONFIG_MASTER_WRITE_DELAY AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated) AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated) AXI_CONFIG_ENABLE_ALL_ASSERTIONS AXI_CONFIG_ENABLE_ASSERTION AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY AXI_CONFIG_READ_DATA_REORDERING_DEPTH AXI_CONFIG_SLAVE_START_ADDR AXI_CONFIG_SLAVE_END_ADDR AXI_CONFIG_MASTER_ERROR_POSITION AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS (AXI4) Configuration name: AXI4_CONFIG_SETUP_TIME AXI4_CONFIG_HOLD_TIME AXI4_CONFIG_BURST_TIMEOUT_FACTOR AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR AXI4_CONFIG_ENABLE_RLAST AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE AXI4_CONFIG_ENABLE_ALL_ASSERTIONS AXI4_CONFIG_ENABLE_ASSERTION AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY AXI4_CONFIG_ENABLE_QOS AXI4_CONFIG_READ_DATA_REORDERING_DEPTH AXI4_CONFIG_SLAVE_START_ADDR AXI4_CONFIG_SLAVE_END_ADDR
config_val	Refer to “Monitor BFM Configuration” on page 510 for description and valid values.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
set_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi_tr_if_0(bfm_index));
set_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi_tr_if_0(bfm_index));
```

AXI4 Example

```
set_config(AXI4_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, 1, bfm_index,
      axi4_tr_if_0(bfm_index));
set_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, 1000, bfm_index,
      axi4_tr_if_0(bfm_index));
```

get_config()

This nonblocking procedure gets the configuration of the monitor BFM.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_config
(
   config_name   : in std_logic_vector(7 downto 0);
   config_val    : out std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0)|integer;
   bfm_id        : in integer;
   path_id       : in *_path_t; --optional
   signal tr_if  : inout *_vhd_if_struct_t
);
```

Arguments

config_name (AXI3) Configuration name:

- AXI_CONFIG_SETUP_TIME
- AXI_CONFIG_HOLD_TIME
- AXI_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI_CONFIG_TIMEOUT_MAX_DATA_TRANSFER
- AXI_CONFIG_BURST_TIMEOUT_FACTOR
- AXI_CONFIG_WRITE_CTRL_TO_DATA_MINTIME
- AXI_CONFIG_WRITE_DATA_TO_CTRL_MINTIME
- AXI_CONFIG_MASTER_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_SLAVE_DEFAULT_UNDER_RESET (deprecated)
- AXI_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI_CONFIG_ENABLE_ASSERTION
- AXI_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI_CONFIG_SLAVE_START_ADDR
- AXI_CONFIG_SLAVE_END_ADDR
- AXI_CONFIG_MASTER_ERROR_POSITION
- AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS

(AXI4) Configuration name:
- AXI4_CONFIG_SETUP_TIME
- AXI4_CONFIG_HOLD_TIME
- AXI4_CONFIG_BURST_TIMEOUT_FACTOR
- AXI4_CONFIG_MAX_TRANSACTION_TIME_FACTOR
- AXI4_CONFIG_ENABLE_RLAST
- AXI4_CONFIG_ENABLE_SLAVE_EXCLUSIVE
- AXI4_CONFIG_ENABLE_ALL_ASSERTIONS
- AXI4_CONFIG_ENABLE_ASSERTION
- AXI4_CONFIG_MAX_LATENCY_AWVALID_ASSERTION_TO_AWREADY
- AXI4_CONFIG_MAX_LATENCY_ARVALID_ASSERTION_TO_ARREADY
- AXI4_CONFIG_MAX_LATENCY_RVALID_ASSERTION_TO_RREADY
- AXI4_CONFIG_MAX_LATENCY_BVALID_ASSERTION_TO_BREADY
- AXI4_CONFIG_MAX_LATENCY_WVALID_ASSERTION_TO_WREADY
- AXI4_CONFIG_ENABLE_QOS
- AXI4_CONFIG_READ_DATA_REORDERING_DEPTH
- AXI4_CONFIG_SLAVE_START_ADDR
- AXI4_CONFIG_SLAVE_END_ADDR

config_val — Refer to "Monitor BFM Configuration" on page 510 for description and valid values.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

- **_PATH_0
- **_PATH_1
- **_PATH_2
- **_PATH_3
- **_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns config_val

AXI3 Example

```
get_config(AXI_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
get_config(AXI_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
     axi_tr_if_0(bfm_index));
```

AXI4 Example

```
get_config(AXI4_CONFIG_SUPPORT_EXCLUSIVE_ACCESS, config_value,
     bfm_index, axi4_tr_if_0(bfm_index));
get_config(AXI4_CONFIG_BURST_TIMEOUT_FACTOR, config_value, bfm_index,
     axi4_tr_if_0(bfm_index));
```

create_monitor_transaction()

This nonblocking procedure creates a monitor transaction. All transaction fields default to legal protocol values, unless previously assigned a value. It returns with the *transaction_id* argument.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure create_monitor_transaction
(
   transaction_id  : out integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Transaction Fields

addr	Start address
size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
burst	Burst type: **_FIXED; **_INCR; (default) **_WRAP; **_BURST_RSVD;
lock	Burst lock: **_NORMAL; (default) **_EXCLUSIVE; **_LOCKED; **_LOCK_RSVD;

Transaction Fields

cache	(AXI3) Burst cache: AXI_NONCACHE_NONBUF; (default) AXI_BUF_ONLY; AXI_CACHE_NOALLOC; AXI_CACHE_BUF_NOALLOC; AXI_CACHE_RSVD0; AXI_CACHE_RSVD1; AXI_CACHE_WTHROUGH_ALLOC_R_ONLY; AXI_CACHE_WBACK_ALLOC_R_ONLY; AXI_CACHE_RSVD2; AXI_CACHE_RSVD3; AXI_CACHE_WTHROUGH_ALLOC_W_ONLY; AXI_CACHE_WBACK_ALLOC_W_ONLY; AXI_CACHE_RSVD4; AXI_CACHE_RSVD5; AXI_CACHE_WTHROUGH_ALLOC_RW; AXI_CACHE_WBACK_ALLOC_RW; (AXI4) Burst cache: AXI4_NONMODIFIABLE_NONBUF; (default) AXI4_BUF_ONLY; AXI4_CACHE_NOALLOC; AXI4_CACHE_2; AXI4_CACHE_3; AXI4_CACHE_RSVD4; AXI4_CACHE_RSVD5; AXI4_CACHE_6; AXI4_CACHE_7; AXI4_CACHE_RSVD8; AXI4_CACHE_RSVD9; AXI4_CACHE_10; AXI4_CACHE_11; AXI4_CACHE_RSVD12; AXI4_CACHE_RSVD12; AXI4_CACHE_14; AXI4_CACHE_15;
prot	Protection: **_NORM_SEC_DATA; (default) **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
id	Burst ID
burst_length	(Optional) Burst length. Default: 0.
data_words	Data words array.
write_strobes	Write strobes array: Each element 0 or 1.
resp	Response: **_OKAY; **_EXOKAY; **_SLVERR; **_DECERR;
region	(AXI4) Region identifier.

Transaction Fields	qos	(AXI4) Quality-of-Service identifier.
	addr_user	Address channel user data.
	data_user	(AXI4) Data channel user data.
	resp_user	(AXI4) Response channel user data.
	read_or_write	Read or write transaction flag: **_TRANS_READ; **_TRANS_WRITE
Operational Transaction Fields	gen_write_strobes	Correction of write strobes for invalid byte lanes: 0 = write_strobes passed through to protocol signals. 1 = write_strobes auto-corrected for invalid byte lanes (default).
	operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING (default);
	delay_mode	Delay mode: **_VALID2READY (default); **_TRANS2READY;
	write_data_mode	Write data mode: **_DATA_AFTER_ADDRESS; The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation. **_DATA_WITH_ADDRESS; (default) The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.
	address_valid_delay	Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.
	data_valid_delay	Data channel RVALID/WVALID delay measured in ACLK cycles for this transaction. Default: 0.
	write_response_valid_delay	Write response channel BVALID delay measured in ACLK cycles for this transaction. Default: 0).
	address_ready_delay	Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction. Default: 0.

	data_ready_delay	Data channel RREADY/WREADY delay measured in ACLK cycles for this transaction. Default: 0.
	write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction. Default: 0.
	data_beat_done	Read data channel phase (beat) *done* array for this transaction.
	transaction_done	Transaction *done* flag for this transaction
Returns	transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221.

AXI3 Example

```
-- Create a monitortransaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_3(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction
-- Returns the transaction ID (tr_id) for this created transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_3(bfm_index));
```

set_addr()

This nonblocking procedure sets the start address *addr* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_addr
(
   addr : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address of transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_addr()

This nonblocking procedure gets the start address *addr* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_addr
(
   addr : out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr	Start address of transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

addr	Start address of transaction.

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the start address addr of the tr_id transaction
get_addr(addr, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_size()

This nonblocking procedure sets the burst *size* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_size
(
   size : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

size	Burst size. Default: width of bus: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_size()

This nonblocking procedure gets the burst *size* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_size
(
   size : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

size	Burst size: **_BYTES_1; **_BYTES_2; **_BYTES_4; **_BYTES_8; **_BYTES_16; **_BYTES_32; **_BYTES_64; **_BYTES_128;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns size

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst size of the tr_id transaction.
get_size (size, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst size of the tr_id transaction.
get_size (size, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_burst()

This nonblocking procedure sets the *burst* type field for a transaction that is uniquely identified by the *transaction_id* field previously created by the create_monitor_transaction() procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_burst
(
   burst: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst	Burst type: **_FIXED; **_INCR (default); **_WRAP; **_BURST_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_burst()

This nonblocking procedure gets the *burst* type field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_burst
(
   burst: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst	Burst type: **_FIXED; **_INCR; **_WRAP; **_BURST_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns burst

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst type of the tr_id transaction.
get_burst (burst, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst type of the tr_id transaction.
get_burst (burst, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_lock()

This nonblocking procedure sets the *lock* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_lock
(
   lock : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

lock	Burst lock: **_NORMAL (default); **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_lock()

This nonblocking procedure gets the *lock* field for a transaction that is uniquely identified by the transaction_id field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_lock
(
   lock : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

lock	Burst lock: **_NORMAL; **_EXCLUSIVE; (AXI3) AXI_LOCKED; (AXI3) AXI_LOCK_RSVD;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns lock

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the lock field of the tr_id transaction.
get_lock(lock, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the lock field of the tr_id transaction.
get_lock(lock, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_cache()

This nonblocking procedure sets the *cache* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_cache
(
   cache: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

cache

(AXI3) Burst cache:
- AXI_NONCACHE_NONBUF; (default)
- AXI_BUF_ONLY;
- AXI_CACHE_NOALLOC;
- AXI_CACHE_BUF_NOALLOC;
- AXI_CACHE_RSVD0;
- AXI_CACHE_RSVD1;
- AXI_CACHE_WTHROUGH_ALLOC_R_ONLY;
- AXI_CACHE_WBACK_ALLOC_R_ONLY;
- AXI_CACHE_RSVD2;
- AXI_CACHE_RSVD3;
- AXI_CACHE_WTHROUGH_ALLOC_W_ONLY;
- AXI_CACHE_WBACK_ALLOC_W_ONLY;
- AXI_CACHE_RSVD4;
- AXI_CACHE_RSVD5;
- AXI_CACHE_WTHROUGH_ALLOC_RW;
- AXI_CACHE_WBACK_ALLOC_RW;

(AXI4) Burst cache:
- AXI4_NONMODIFIABLE_NONBUF; (default)
- AXI4_BUF_ONLY;
- AXI4_CACHE_NOALLOC;
- AXI4_CACHE_2;
- AXI4_CACHE_3;
- AXI4_CACHE_RSVD4;
- AXI4_CACHE_RSVD5;
- AXI4_CACHE_6;
- AXI4_CACHE_7;
- AXI4_CACHE_RSVD8;
- AXI4_CACHE_RSVD9;
- AXI4_CACHE_10;
- AXI4_CACHE_11;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_14;
- AXI4_CACHE_15;

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

Note

You do not normally use this procedure in a monitor test program.

get_cache()

This nonblocking procedure gets the *cache* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_cache
(
   cache: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

cache

(AXI3) Burst cache:
- AXI_NONCACHE_NONBUF; (default)
- AXI_BUF_ONLY;
- AXI_CACHE_NOALLOC;
- AXI_CACHE_BUF_NOALLOC;
- AXI_CACHE_RSVD0;
- AXI_CACHE_RSVD1;
- AXI_CACHE_WTHROUGH_ALLOC_R_ONLY;
- AXI_CACHE_WBACK_ALLOC_R_ONLY;
- AXI_CACHE_RSVD2;
- AXI_CACHE_RSVD3;
- AXI_CACHE_WTHROUGH_ALLOC_W_ONLY;
- AXI_CACHE_WBACK_ALLOC_W_ONLY;
- AXI_CACHE_RSVD4;
- AXI_CACHE_RSVD5;
- AXI_CACHE_WTHROUGH_ALLOC_RW;
- AXI_CACHE_WBACK_ALLOC_RW;

(AXI4) Burst cache:
- AXI4_NONMODIFIABLE_NONBUF; (default)
- AXI4_BUF_ONLY;
- AXI4_CACHE_NOALLOC;
- AXI4_CACHE_2;
- AXI4_CACHE_3;
- AXI4_CACHE_RSVD4;
- AXI4_CACHE_RSVD5;
- AXI4_CACHE_6;
- AXI4_CACHE_7;
- AXI4_CACHE_RSVD8;
- AXI4_CACHE_RSVD9;
- AXI4_CACHE_10;
- AXI4_CACHE_11;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_RSVD12;
- AXI4_CACHE_14;
- AXI4_CACHE_15;

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

	path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
	tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
Returns	cache	

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the cache field of the tr_id transaction.
get_cache(cache, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the cache field of the tr_id transaction.
get_cache(cache, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_prot()

This nonblocking procedure sets the protection *prot* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_prot
(
   prot: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

prot	Burst protection: **_NORM_SEC_DATA (default); **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_prot()

This nonblocking procedure gets the protection *prot* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_prot
(
   prot: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

prot	Burst protection: **_NORM_SEC_DATA; **_PRIV_SEC_DATA; **_NORM_NONSEC_DATA; **_PRIV_NONSEC_DATA; **_NORM_SEC_INST; **_PRIV_SEC_INST; **_NORM_NONSEC_INST; **_PRIV_NONSEC_INST;
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

prot

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the protection field of the tr_id transaction.
get_prot(prot, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_id()

This nonblocking procedure sets the *id* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_id
(
   id: in integer;
   transaction_id  : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0) | integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

id	Burst ID
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_id()

This nonblocking procedure gets the *id* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_id
(
   id: out integer;
   transaction_id  : in std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0) | integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

id	Burst ID
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the id field of the tr_id transaction.
get_id(id, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_burst_length()

This nonblocking procedure sets the *burst_length* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Note

The *burst_length* field is the value that appears on the AWLEN and the ARLEN protocol signals. The number of data phases (beats) in a data burst is therefore *burst_length* + 1.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_burst_length
(
   burst_length : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0)
   | integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst_length	Burst length (default = 0).
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_burst_length()

This nonblocking procedure gets the *burst_length* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Note

The *burst_length* field is the value that appears on the AWLEN and the ARLEN protocol signals. The number of data phases (beats) in a data burst is therefore *burst_length* + 1.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_burst_length
(
   burst_length : out std_logic_vector(**_MAX_BIT_SIZE-1 downto
   0) | integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

burst_length	Burst length.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns burst_length

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the burst length field of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the burst length field of the tr_id transaction.
get_burst_length(burst_length, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_words()

This nonblocking procedure sets the *data_words* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_data_words
(
   data_words: in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_words	Data words array.
index	(Optional) Array element number for *data_words*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_data_words()

This nonblocking procedure gets a *data_words* field array element for a transaction that is uniquely identified by the *transactionid* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_data_words
(
   data_words: out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_words	Data words array.
index	(Optional) Array element number for *data_words*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

data_words

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the data_words field of the first data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the data_words field of the second data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the data_words field of the first data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the data_words field of the second data phase (beat)
-- for the tr_id transaction.
get_data_words(data, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_write_strobes()

This nonblocking procedure sets the *write_strobes* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure and uniquely identified by the *transaction_id* field.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_write_strobes
(
   write_strobes : in std_logic_vector (**_MAX_BIT_SIZE-1 downto
   0) | integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_strobes	Write strobes array.
index	(Optional) Array element number for *write_strobes*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_write_strobes()

This nonblocking procedure gets the *write_strobes* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_write_strobes
(
   write_strobes : out std_logic_vector (**_MAX_BIT_SIZE-1 downto
   0) | integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_strobes	Write strobes array.
index	(Optional) Array element number for *write_strobes*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

write_strobes

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the write_strobes field of the first data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the write_strobes field of the second data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the write_strobes field of the first data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the write_strobes field of the second data phase (beat)
-- for the tr_id transaction.
get_write_strobes(write_strobe, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_resp()

This nonblocking procedure sets the response *resp* field array elements for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_resp
(
   resp: in std_logic_vector (**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

resp	Transaction response array: **_OKAY = 0; **_EXOKAY = 1; **_SLVERR = 2; **_DECERR = 3;
index	(Optional) Array element number for *resp*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_resp()

This nonblocking procedure gets a response *resp* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_resp
(
   resp: out std_logic_vector (**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

resp — Transaction response array:

**_OKAY = 0;
**_EXOKAY = 1;
**_SLVERR = 2;
**_DECERR = 3;

index — (Optional) Array element number for *resp*.

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

resp

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the response field of the first data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 0, tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the response field of the second data phase (beat)
-- if the tr_id transaction.
get_resp(read_resp, 1, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the response field of the first data phase (beat)
-- of the tr_id transaction.
get_resp(read_resp, 0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the response field of the second data phase (beat)
-- if the tr_id transaction.
get_resp(read_resp, 1, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_addr_user()

This nonblocking procedure sets the user data *addr_user* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_addr_user
(
   addr_user : in std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr_user	User data in address phase.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_addr_user()

This nonblocking procedure gets the user data *addr_user* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_addr_user
(
   addr_user : out std_logic_vector(**_MAX_BIT_SIZE-1 downto 0) |
   integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

addr_user	User data in the address phase.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns addr_user

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel user data of the tr_id transaction.
get_addr_user(user_data, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel user data of the tr_id transaction.
get_addr_user(user_data, tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

set_read_or_write()

This procedure sets the *read_or_write* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the create_monitor_transaction() procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_read_or_write
(
   read_or_write: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

read_or_write	Read or write transaction: **_TRANS_READ = 0 **_TRANS_WRITE = 1
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_read_or_write()

This nonblocking procedure gets the *read_or_write* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_read_or_write
(
   read_or_write: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

read_or_write	Read or write transaction: **_TRANS_READ = 0 **_TRANS_WRITE = 1
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

read_or_write

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read_or_write field of tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read_or_write field of tr_id transaction.
get_read_or_write(read_or_write, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_gen_write_strobes()

This nonblocking procedure sets the *gen_write_strobes* field for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_gen_write_strobes
(
   gen_write_strobes: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

gen_write_strobes	Correction of write strobes for invalid byte lanes: 0 = *write_strobes* passed through to protocol signals. 1 = *write_strobes* auto-corrected for invalid byte lanes (default).
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_gen_write_strobes()

This nonblocking procedure gets the *gen_write_strobes* field for a write transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_gen_write_strobes
(
   gen_write_strobes: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

gen_write_strobes	Correct write strobes flag: 0 = *write_strobes* passed through to protocol signals. 1 = *write_strobes* auto-corrected for invalid byte lanes.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns gen_write_strobes

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify the
transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify the
transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the auto correction write strobes flag of the tr_id transaction.
get_gen_write_strobes(write_strobes_flag, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_operation_mode()

This nonblocking procedure sets the *operation_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_operation_mode
(
   operation_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING (default);
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Set the operation mode to nonblocking for the tr_id transaction.
set_operation_mode(AXI_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Set the operation mode to nonblocking for the tr_id transaction.
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

get_operation_mode()

This nonblocking procedure gets the *operation_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_operation_mode
(
   operation_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

operation_mode	Operation mode: **_TRANSACTION_NON_BLOCKING; **_TRANSACTION_BLOCKING;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns operation_mode

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the operation mode of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the operation mode of the tr_id transaction.
get_operation_mode(operation_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_delay_mode()

This AXI3 nonblocking procedure sets the *delay_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_delay_mode
(
   delay_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

delay_mode	Delay mode: AXI_VALID2READY (default); AXI_TRANS2READY;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_delay_mode()

This AXI3 nonblocking procedure gets the *delay_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_delay_mode
(
   delay_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

delay_mode	Delay mode: AXI_VALID2READY; AXI_TRANS2READY;
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns delay_mode

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
....
-- Get the delay mode of the *VALID to *READY handshake of the
-- tr_id transaction
get_delay_mode(delay_mode, tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 BFM

Note

This procedure is not supported in the AXI4 BFM API.

set_write_data_mode()

This nonblocking procedure sets the *write_data_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_write_data_mode
(
   write_data_mode: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_data_mode — Write data mode:

**_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation.

**_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id — (Optional) Parallel process path identifier:
**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4
Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_write_data_mode()

This nonblocking procedure gets the *write_data_mode* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_write_data_mode
(
   write_data_mode: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_data_mode — Write data mode:

**_DATA_AFTER_ADDRESS;
The master first drives the address phase and, after it completes, it drives the corresponding data phases. The master waits for AWREADY before asserting WVALID. For a slave designed to wait for WVALID before asserting AWREADY, using this mode may cause a deadlock situation. This mode will force the data transfer to start after the address transfer completes; however, it is recommended that you use the **_DATA_WITH_ADDRESS along with a *data_valid_delay* setting instead to avoid the possible deadlock situation.

**_DATA_WITH_ADDRESS; (default)
The master drives the address and the data phase in a nonblocking process; it asserts AWVALID and then asserts WVALID depending on *data_valid_delay*. If *data_valid_delay* is set to 0, then AWVALID and WVALID are asserted at the same time; otherwise, WVALID is asserted after *data_valid_delay*.

transaction_id — Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

bfm_id — BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

tr_if — Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

write_data_mode

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data mode of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data mode of the tr_id transaction
get_write_data_mode(write_data_mode, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_address_valid_delay()

This nonblocking procedure sets the *address_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_address_valid_delay
(
   address_valid_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_valid_delay	Address channel ARVALID/AWVALID delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_address_valid_delay()

This nonblocking procedure gets the *address_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_address_valid_delay
(
   address_valid_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_valid_delay	Address channel ARVALID/AWVALID delay in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns address_valid_delay

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel delay of the tr_id transaction.
get_address_valid_delay(address_valid_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_address_ready_delay()

This AXI3 nonblocking procedure sets the *address_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedures.

Prototype

```
set_address_ready_delay
(
   address_ready_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

address_ready_delay	Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_address_ready_delay()

This nonblocking procedure gets the *address_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_address_ready_delay
(
   address_ready_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

address_ready_delay	Address channel ARREADY/AWREADY delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns address_ready_delay

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the address channel *READY delay of the tr_id transaction.
get_address_ready_delay(address_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_valid_delay()

This nonblocking procedure sets the *data_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
set_data_valid_delay
(
   data_valid_delay: in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_valid_delay	Data channel array to hold RVALID/WVALID delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element number for *data_valid_delay*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_data_valid_delay()

This nonblocking procedure sets the *data_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_data_valid_delay
(
   data_valid_delay: out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_valid_delay	Data channel array to hold RVALID/WVALID delays measured in ACLK cycles for this transaction.
index	(Optional) Array element number for *data_valid_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

data_valid_delay

AXI3 Example

```
-- Create a monitor transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....
-- Get the write channel WVALID delay for the first data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the write channel WVALID delay for the second data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction with start address of 0.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....
-- Get the write channel WVALID delay for the first data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the write channel WVALID delay for the second data
-- phase (beat) of the tr_id transaction.
get_data_valid_delay(data_valid_delay, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_ready_delay()

This AXI3 nonblocking procedure sets the *data_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
 set_data_ready_delay
(
   data_ready_delay: in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

data_ready_delay	Data channel array to hold RREADY?WREADY delays measured in ACLK cycles for this transaction. Default: 0.
index	(Optional) Array element number for *data_ready_delay*.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_data_ready_delay()

This nonblocking procedure gets the *data_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_data_ready_delay
(
   data_ready_delay: out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_ready_delay	Data channel array to hold RREADY/WREADY delay measured in ACLK cycles for this transaction.
index	(Optional) Array element index number for *data_ready_delay.*
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the first
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the second
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the first
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

-- Get the read data channel RREADY delay for the second
-- data phase (beat) of the tr_id transaction.
get_data_ready_delay(data_ready_delay, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_write_response_valid_delay()

This AXI3 nonblocking procedure sets the *write_response_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
 set_write_response_valid_delay
(
   write_response_valid_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

write_response_valid_delay	Write data channel BVALID delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_write_response_valid_delay()

This nonblocking procedure gets the *write_response_valid_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_write_response_valid_delay
(
   write_response_valid_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_valid_delay	Write data channel BVALID delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

write_response_valid_delay

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id,
bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BVALID delay of the tr_id transaction.
get_write_response_valid_delay(write_response_valid_delay, tr_id,
bfm_index, axi4_tr_if_0(bfm_index));
```

set_write_response_ready_delay()

This AXI3 nonblocking procedure sets the *write_response_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
set_write_response_ready_delay
(
   write_response_ready_delay: in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in axi_path_t; --optional
   signal tr_if : inout axi_vhd_if_struct_t
);
```

Arguments

write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction. Default: 0.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to v for more details.
path_id	(Optional) Parallel process path identifier: AXI_PATH_0 AXI_PATH_1 AXI_PATH_2 AXI_PATH_3 AXI_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_write_response_ready_delay()

This nonblocking procedure gets the *write_response_ready_delay* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_write_response_ready_delay
(
   write_response_ready_delay: out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

write_response_ready_delay	Write data channel BREADY delay measured in ACLK cycles for this transaction.
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns write_response_ready_delay

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response channel BREADY delay of the tr_id transaction.
get_write_response_ready_delay(write_resp_ready_delay, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_data_beat_done()

This nonblocking procedure sets the *data_beat_done* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_data_beat_done
(
   data_beat_done : in integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_beat_done	Write data channel phase (beat) *done* array for this transaction.
index	(Optional) Array element number for *data_beat_done*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_data_beat_done()

This nonblocking procedure gets the *data_beat_done* field array element for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
get_data_beat_done
(
   data_beat_done : out integer;
   index : in integer; --optional
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

data_beat_done	Data channel phase (beat) *done* array for this transaction
index	(Optional) Array element number for *data_beat_done*.
transaction_id	Transaction identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns

data_beat_done

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 0, tr_id, bfm_index,
axi_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 1, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the first
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 0, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));

....
-- Get the read data channel data_beat_done flag for the second
-- data phase (beat) of the tr_id transaction.
get_data_beat_done(data_beat_done, 1, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

set_transaction_done()

This nonblocking procedure sets the *transaction_done* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedures.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
set_transaction_done
(
   transaction_done : in integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

Note

You do not normally use this procedure in a monitor test program.

get_transaction_done()

This nonblocking procedure gets the *transaction_done* field for a transaction that is uniquely identified by the *transaction_id* field previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
get_transaction_done
(
   transaction_done : out integer;
   transaction_id  : in integer;
   bfm_id : in integer;
   path_id : in *_path_t; --optional
   signal tr_if : inout *_vhd_if_struct_t
);
```

Arguments

transaction_done	Transaction *done* flag for this transaction
transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns transaction_done

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the transaction_done flag of the tr_id transaction.
get_transaction_done(transaction_done, tr_id, bfm_index,
axi4_tr_if_0(bfm_index));
```

get_read_data_burst()

This blocking procedure gets a read data burst that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

It calls the *get_read_data_phase()* procedure for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_read_data_burst
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read data burst for the tr_id transaction.
get_read_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(0, tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read data burst for the tr_id transaction.
get_read_data_burst(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_data_phase()

This blocking procedure gets a read data phase that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

The *get_read_data_phase()* sets the *data_beat_done* array *index* element field to 1 when the phase completes. If this is the last phase (beat) of the burst, then it sets the *transaction_done* field to 1 to indicate the whole read transaction has completed.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_read_data_phase
(
   transaction_id  : in integer;
   index : in integer; --optional
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
index	(Optional) Data phase (beat) number.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read data phase for the first beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 0, bfm_index, axi_tr_if_0(bfm_index));

-- Get the read data phase for the second beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 1, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read data phase for the first beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 0, bfm_index, axi4_tr_if_0(bfm_index));

-- Get the read data phase for the second beat of the
-- tr_id transaction.
get_read_data_phase(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_response_phase()

This blocking procedure gets a write response phase that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

It sets the *transaction_done* field to 1 when the phase completes to indicate the whole transaction has completed.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure get_write_response_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write response phase for the tr_id transaction.
get_write_response_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction.
-- Creation returns tr_id to identify the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write response phase for the tr_id transaction.
get_write_response_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_addr_phase()

This blocking procedure gets a write address phase that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_write_addr_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; -- Optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write address phase of the tr_id transaction.
get_write_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write address phase of the tr_id transaction.
get_write_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_addr_phase()

This blocking procedure gets a read address phase that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_read_addr_phase
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; -- Optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the read address phase of the tr_id transaction.
get_read_addr_phase(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the read address phase of the tr_id transaction.
get_read_addr_phase(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_data_phase()

This blocking procedure gets a write data phase that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure. The *get_write_data_phase()* sets the *data_beat_done* array *index* element to 1 when the phase completes. If this is the last data phase of the burst, then it returns the *last* argument set to 1.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_write_data_phase
(
   transaction_id  : in integer;
   index : in integer; --optional
   last : out integer;
   bfm_id          : in integer;
   path_id          : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
index	(Optional) Data phase (beat) number.
last	Last data phase (beat) of the burst: 0 = data burst not complete 1 = data burst complete
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

last

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data phase for the first beat of the tr_id transaction.
get_write_data_phase(tr_id, 0, last, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data phase for the second beat of the tr_id transaction.
get_write_data_phase(tr_id, 1, last, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data phase for the first beat of the tr_id transaction.
get_write_data_phase(tr_id, 0, last, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the write data phase for the second beat of the tr_id transaction.
get_write_data_phase(tr_id, 1, last, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_data_burst()

This blocking procedure gets a write data burst that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

It calls the *get_write_data_phase()* procedure for each beat of the data burst, with the length of the burst defined by the transaction record *burst_length* field.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure get_write_data_burst
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data burst for the tr_id transaction.
get_write_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the write data burst for the tr_id transaction.
get_write_data_burst(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

get_rw_transaction()

This blocking procedure gets a complete read/write transaction that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure get_rw_transaction
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Get the complete tr_id transaction.
get_rw_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Get the complete tr_id transaction.
get_rw_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_addr_ready()

This blocking AXI4 procedure returns the value of the read address channel ARREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_read_addr_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the ARREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_read_addr_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the ARREADY signal value
bfm.get_read_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_read_data_ready()

This blocking AXI4 procedure returns the value of the read data channel RREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_read_data_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the RREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_read_data_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the RREADY signal value
bfm.get_read_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_addr_ready()

This blocking AXI4 procedure returns the value of the write address channel AWREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_write_addr_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the AWREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_write_addr_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the WREADY signal value
bfm.get_write_addr_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_data_ready()

This blocking AXI4 procedure returns the value of the write data channel WREADY signal using the *ready* argument. It will block for one ACLK period.

Prototype

```
procedure get_write_data_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the WREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_write_data_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the WREADY signal value
bfm.get_write_data_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

get_write_resp_ready()

This blocking AXI4 procedure returns the value of the write response channel BREADY signal using the *ready* argument. It blocks for one ACLK period.

Prototype

```
procedure get_write_resp_ready
(
   ready : out integer;
   bfm_id          : in integer;
   path_id         : in axi4_adv_path_t; --optional
   signal tr_if    : inout axi4_vhd_if_struct_t
);
```

Arguments

ready	The value of the RREADY signal.
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: AXI4_PATH_5 AXI4_PATH_6 AXI4_PATH_7 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns ready

AXI3 BFM

Note

The *get_write_resp_ready()* procedure is not available in the AXI3 BFM.

AXI4 Example

```
// Get the BREADY signal value
bfm.get_write_resp_ready(ready, bfm_index, axi4_tr_if_0(bfm_index));
```

push_transaction_id()

This nonblocking procedure pushes a transaction record into the back of a queue. The transaction is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure. The queue is identified by the *queue_id* argument.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure push_transaction_id
(
   transaction_id  : in integer;
   queue_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

queue_id — Queue identifier:

**_QUEUE_ID_0
**_QUEUE_ID_1
**_QUEUE_ID_2
**_QUEUE_ID_3
**_QUEUE_ID_4
AXI4_QUEUE_ID_5
AXI4_QUEUE_ID_6
AXI4_QUEUE_ID_7

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

**_PATH_0
**_PATH_1
**_PATH_2
**_PATH_3
**_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI_QUEUE_ID_1, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Push the transaction record into queue 1 for the tr_id transaction.
push_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
```

pop_transaction_id()

This nonblocking (unless queue is empty) procedure pops a transaction record from the front of a queue. The transaction is uniquely identified by the *transaction_id* argument previously created by the *get_rw_transaction()* procedure. The queue is identified by the *queue_id* argument.

If the queue is empty, then it will block until an entry becomes available.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure pop_transaction_id
(
   transaction_id  : in integer;
   queue_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id — Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

queue_id — Queue identifier:

- **_QUEUE_ID_0
- **_QUEUE_ID_1
- **_QUEUE_ID_2
- **_QUEUE_ID_3
- **_QUEUE_ID_4
- AXI4_QUEUE_ID_5
- AXI4_QUEUE_ID_6
- AXI4_QUEUE_ID_7

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

bfm_id — BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

path_id — (Optional) Parallel process path identifier:

- **_PATH_0
- **_PATH_1
- **_PATH_2
- **_PATH_3
- **_PATH_4

Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

tr_if — Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns

None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI_QUEUE_ID_1, bfm_index,
axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Pop the transaction record from queue 1 for the tr_id transaction.
pop_transaction_id(tr_id, AXI4_QUEUE_ID_1, bfm_index,
axi4_tr_if_0(bfm_index));
```

print()

This nonblocking procedure prints a transaction record, that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure print
(
   transaction_id  : in integer;
   print_delays : in integer; --optional
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
print_delays	(Optional) Print delay values flag: 0 = do not print the delay values (default). 1 = print the delay values.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Print the transaction record (including delay values) of the
-- tr_id transaction.
print(tr_id, 1, bfm_index, axi4_tr_if_0(bfm_index));
```

destruct_transaction()

This blocking procedure removes a transaction record, for cleanup purposes and memory management, that is uniquely identified by the *transaction_id* argument previously created by the *create_monitor_transaction()* procedure.

Prototype

```
-- * = axi | axi4
-- ** = AXI | AXI4
procedure destruct_transaction
(
   transaction_id  : in integer;
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

transaction_id	Transaction identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
bfm_id	BFM identifier. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.
tr_if	Transaction signal interface. Refer to "Overloaded Procedure Common Arguments" on page 221 for more details.

Returns None

AXI3 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
-- Create a monitor transaction. Creation returns tr_id to identify
-- the transaction.
create_monitor_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));

....

-- Remove the transaction record for the tr_id transaction.
destruct_transaction(tr_id, bfm_index, axi4_tr_if_0(bfm_index));
```

wait_on()

This blocking procedure waits for an event on the ACLK or ARESETn signals to occur before proceeding. An optional *count* argument waits for the number of events equal to *count.*

Prototype

```
-- * = axi| axi4
-- ** = AXI | AXI4
procedure wait_on
(
   phase           : in integer;
   count: in integer; --optional
   bfm_id          : in integer;
   path_id         : in *_path_t; --optional
   signal tr_if    : inout *_vhd_if_struct_t
);
```

Arguments

phase	Wait for: **_CLOCK_POSEDGE **_CLOCK_NEGEDGE **_CLOCK_ANYEDGE **_CLOCK_0_TO_1 **_CLOCK_1_TO_0 **_RESET_POSEDGE **_RESET_NEGEDGE **_RESET_ANYEDGE **_RESET_0_TO_1 **_RESET_1_TO_0
count	(Optional) Wait for a number of events to occur set by *count*. (default = 1)
bfm_id	BFM identifier. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
path_id	(Optional) Parallel process path identifier: **_PATH_0 **_PATH_1 **_PATH_2 **_PATH_3 **_PATH_4 Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.
tr_if	Transaction signal interface. Refer to “Overloaded Procedure Common Arguments” on page 221 for more details.

Returns None

AXI3 Example

```
wait_on(AXI_RESET_POSEDGE, bfm_index, axi_tr_if_0(bfm_index));
wait_on(AXI_CLOCK_POSEDGE, 10, bfm_index, axi_tr_if_0(bfm_index));
```

AXI4 Example

```
wait_on(AXI4_RESET_POSEDGE, bfm_index, axi4_tr_if_0(bfm_index));
wait_on(AXI4_CLOCK_POSEDGE, 10, bfm_index, axi4_tr_if_0(bfm_index));
```

Chapter 11
VHDL Tutorials

This chapter discusses how to use the Mentor VIP – Intel FPGA Edition master and slave BFMs to verify slave and master components, respectively.

In the Verifying a Slave DUT tutorial, the slave is an on-chip RAM model that is verified using a master BFM and test program. In the Verifying a Master DUT tutorial, the master issues simple write and read transactions that are verified using a slave BFM and test program.

Following this top-level discussion of how you verify a master and a slave component using the Mentor VIP – Intel FPGA Edition is a brief example of how to run Qsys, the powerful system integration tool in the Quartus Prime software. This procedure shows you how to use Qsys to create a top-level DUT environment. For more details on this example, refer to "Getting Started with Qsys and the BFMs" on page 673.

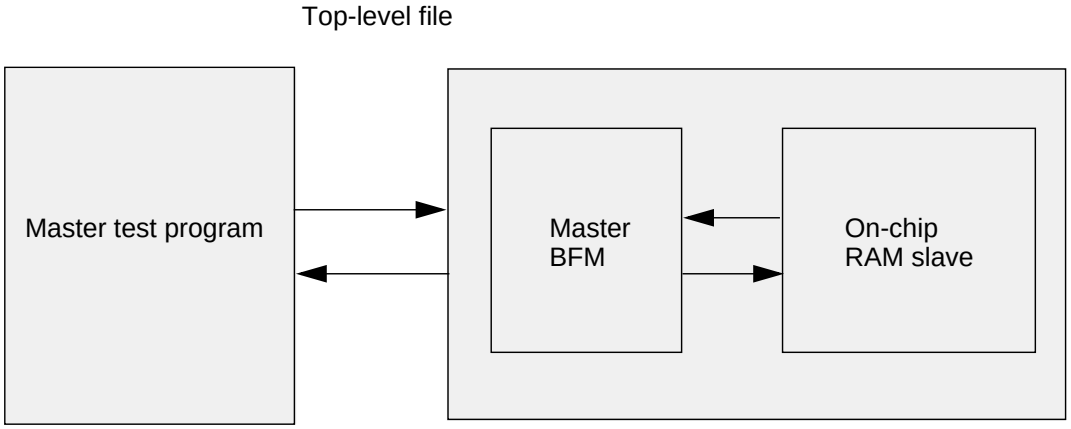
Note

Parameters to configure any optional signals, master BFM transaction issuing, and slave BFM acceptance capability, are set with the Qsys Parameter Editor. See "Running the Qsys Tool" on page 676 for details of the Qsys Parameter Editor.

Verifying a Slave DUT

A slave DUT component is connected to a master BFM at the signal-level. A master test program, written at the transaction-level, generates stimulus via the master BFM to verify the slave DUT. Figure 11-1 illustrates a typical top-level test bench environment.

Figure 11-1. Slave DUT Top-Level Test Bench Environment



In this example, the master test program also compares the written data with that read back from the slave DUT, reporting the result of the comparison.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

AXI3 BFM Master Test Program

Using the AXI3 master BFM API, this master test program creates a wide range of stimulus scenarios that test the slave DUT. This tutorial restricts the stimulus to a write transaction followed by a read transaction to the same address, which then compares the read data with the previously written data.

Note

For a complete code example of this master test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi3\qsys-examples\ex1_back_to_back_vhd\master_test_program.vhd

Configuration and Initialization

The code excerpt in Example 11-1 shows the master test program architecture definition *master_test_program_a*. It defines three variables, *tr_id*, *data_words*, and *lp* to hold the transaction identifier number, data words payload, and read data from the slave, respectively. An additional system clock cycle is waited on after reset to satisfy the AXI3 protocol requirement specified in Section 11.1.2 of the AMBA AXI Protocol Specification before executing transactions.

Example 11-1. Architecture Definition and Initialization

```
architecture master_test_program_a of master_test_program is
begin

   -- Master test
   process
   variable tr_id: integer;
   variable data_words :  std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
   variable lp: line;

   begin
      wait_on(AXI_RESET_0_TO_1, index, axi_tr_if_0(index));
      wait_on(AXI_CLOCK_POSEDGE, index, axi_tr_if_0(index));
```

Create and Execute Write Transactions

To generate AXI3 protocol traffic, the master test program must create the transaction before executing it. The code excerpt in Example 11-2 uses the VHDL Master API *create_write_transaction()* procedure, providing only the start address argument of the

transaction. The optional burst-length argument automatically defaults to a value of zero, indicating a burst length of a single beat.

This example has an AXI3 data bus width of 32 bits; therefore, a single beat of data conveys 4 bytes across the data bus. The *set_data_words()* procedure sets the *data_words[0]* transaction field with the value of 1 on byte lane 1, resulting in a value of 32'h0000_0100. However, the AXI3 protocol permits narrow transfers with the use of the write strobes signal WSTRB to indicate which byte lane contains valid write data, and therefore indicates to the slave DUT which data byte lane will be written into memory. Similarly, you can call the *set_write_strobes()* procedure to set the *write_strobes[0]* transaction field with the value of 4'b0010, indicating that only valid data is being transferred on byte lane 1. The write transaction then executes on the protocol signals using the *execute_transaction()* procedure.

All other transaction fields default to legal protocol values. Refer to "Master BFM Configuration" on page 222 for details.

Example 11-2. Create and Execute Write Transactions

```
-- Write data value 1 on byte lanes 1 to address 1.
create_write_transaction(1, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"00000100";
set_data_words(data_words, tr_id, index, axi_tr_if_0(index));
set_write_strobes(2, tr_id, index, axi_tr_if_0(index));
report "master_test_program: Writing data (1) to address (1)";

-- By default it will run in Blocking mode
execute_transaction(tr_id, index, axi_tr_if_0(index));
```

In the complete master test program, three subsequent write transactions are created and executed in a similar manner to Example 11-2.

Create and Execute Read Transactions

The code excerpt in Example 11-3 reads the data that has been previously written into the slave memory. The master test program first creates a read transaction using the *create_read_transaction()* procedure, providing only the start address argument of the transaction. The optional burst-length argument automatically defaults to a value of zero, indicating a burst length of a single beat.

The *set_id()* procedure is then called to set the transaction *id* field to be 1, and the *set_size()* procedure sets the transaction *size* field to be a single byte (AXI_BYTES_1). The read transaction is then executed onto the protocol signals with a call to the *execute_transaction()* procedure.

The read data is obtained using the *get_data_words()* procedure to get the *data_words[0]* transaction field value. The result of the read data is compared with the expected data, and a report message displays the transcript.

Example 11-3. Create and Execute Read Transactions

```
--Read data from address 1.
create_read_transaction(1, tr_id, index, axi_tr_if_0(index));
set_id(1, tr_id, index, axi_tr_if_0(index));
set_size(AXI_BYTES_1, tr_id, index, axi_tr_if_0(index));
execute_transaction(tr_id, index, axi_tr_if_0(index));

get_data_words(data_words, tr_id, index, axi_tr_if_0(index));
if(data_words(31 downto 0) = x"00000100") then
      report "master_test_program: Read correct data (1) at address (1)";
else
      hwrite(lp, data_words(31 downto 0));
      report "master_test_program: Error: Expected data (1) at address 1,
      but got " & lp.all;
end if;
```

In the complete master test program, three subsequent read transactions are created and executed in a similar manner to Example 11-3.

Create and Execute Write Burst Transactions

The code excerpt in Example 11-4 calls the *create_write_transaction()* procedure to create a write burst transaction by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 7+1=8.

Note

The burst length argument passed to the *create_write_transaction()* procedure is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the AWLEN protocol signals.

The *set_data_words()* procedure is then called eight times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst; therefore, a *for* loop calls the *set_write_strobes()* procedure to set the *write_strobes* fields of the transaction to 15 for each beat of the burst.

The write transaction is then executed onto the protocol signals.

Example 11-4. Create and Execute Write Burst Transactions

```
--  Write data burst length of 7 to start address 16.
create_write_transaction(16, 7, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE0ACE1";
set_data_words(data_words, 0, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE2ACE3";
set_data_words(data_words, 1, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE4ACE5";
set_data_words(data_words, 2, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE6ACE7";
set_data_words(data_words, 3, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE8ACE9";
set_data_words(data_words, 4, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACEAACEB";
set_data_words(data_words, 5, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACECACED";
set_data_words(data_words, 6, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACEEACEF";
set_data_words(data_words, 7, tr_id, index, axi_tr_if_0(index));
for i in  0 to 7 loop
   set_write_strobes(15, i, tr_id, index, axi_tr_if_0(index));
end loop;
execute_transaction(tr_id, index, axi_tr_if_0(index));
```

In the complete master test program, a subsequent write data burst transaction with a start address of 128 is created and executed in a similar manner to Example 11-4.

Create and Execute Read Burst Transactions

The code excerpt in Example 11-5 reads the first two data beats from the data burst that have been previously written into the slave memory. The call to the *create_read_transaction()* procedure creates the read burst transaction by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 1+1=2.

Note

The burst length argument passed to the *create_read_transaction()* procedure is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the ARLEN protocol signals.

The read transaction is then executed onto the protocol signals by calling the *execute_transaction()* procedure. The read data is obtained by calling the *get_data_words()* procedure twice to get the *data_words* transaction field values. The result of the read data is compared with the expected data, and a message displays the transcript.

Example 11-5. Create and Execute Read Burst Transactions

```
-- Read data burst of length 1 from address 16.
create_read_transaction(16, 1, tr_id, index, axi_tr_if_0(index));
execute_transaction(tr_id, index, axi_tr_if_0(index));

get_data_words(data_words, 0, tr_id, index, axi_tr_if_0(index));
if(data_words(31 downto 0) = x"ACE0ACE1") then
   report "master_test_program: Read correct data (hACE0ACE1) at
           address(16)";
else
   hwrite(lp, data_words(31 downto 0));
   report "master_test_program: Error: Expected data (hACE0ACE1) at
           address (16), but got " & lp.all;
end if;
get_data_words(data_words, 1, tr_id, index, axi_tr_if_0(index));
if(data_words(31 downto 0) = x"ACE2ACE3") then
   report "master_test_program: Read correct data (hACE2ACE3) at
           address (20)";
else
hwrite(lp, data_words(31 downto 0));
report "master_test_program: Error: Expected data (hACE2ACE3) at
         address (20), but got " & lp.all;
end if;
```

In the complete master test program, a subsequent read transaction with a start address of 128 is created and executed in a similar manner to Example 11-5.

Create and Execute Outstanding Write Burst Transactions

The code excerpt in Example 11-2 uses the AXI3 Master BFM *create_write_transaction()* procedure to create a write burst transaction by providing the start address and burst length arguments. The actual length of the burst on the protocol wires is 3+1=4.

Note

The burst length argument passed to the *create_read_transaction()* procedure is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the ARLEN protocol signals.

The *set_data_words()* procedure is then called four times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst, therefore a *for .. loop* uses the *set_write_strobes()* procedure to set the *write_strobes* fields of the transaction to 15.

The call to the *set_operation_mode()* procedure configures the transaction to be nonblocking by setting the *operation_mode* field to AXI_TRANSACTION_NON_BLOCKING.

The write transaction is then executed onto the protocol signals by calling the *execute_transaction()* procedure. The executed transaction will be nonblocking, allowing subsequent address phase transactions to be executed before the current write data burst has completed. This allows outstanding write transaction stimulus to be created.

Example 11-6. Create and Execute Outstanding Write Burst Transactions

```
create_write_transaction(0, 3, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE0ACE1";
set_data_words(data_words, 0, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE2ACE3";
set_data_words(data_words, 1, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE4ACE5";
set_data_words(data_words, 2, tr_id, index, axi_tr_if_0(index));
data_words(31 downto 0) := x"ACE6ACE7";
set_data_words(data_words, 3, tr_id, index, axi_tr_if_0(index));
for i in  0 to 3 loop
   set_write_strobes(15, i, tr_id, index, axi_tr_if_0(index));
end loop;
set_operation_mode(AXI_TRANSACTION_NON_BLOCKING, tr_id, index,
                        axi_tr_if_0(index));
execute_transaction(tr_id, index, axi_tr_if_0(index));
```

In the complete master test program, subsequent write transactions are created and executed in a similar manner to Example 11-2.

AXI4 BFM Master Test Program

A master test program using the master BFM API is capable of creating a wide range of stimulus scenarios to verify a slave DUT. However, this tutorial restricts the master BFM stimulus to write transactions followed by read transactions to the same address, and then compares the read data with the previously written data.

Note

For a complete code example of this master test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi4\qsys-examples\ex1_back_to_back_vhd\master_test_program.vhd

The master test program contains the following:

- A create_transactions process that creates and executes read and write transactions.
- Processes *handle_write_resp_ready* and *handle_read_data_ready* to handle the write response channel BREADY and read data channel RREADY signals, respectively.

- Variables *m_wr_resp_phase_ready_delay* and *m_rd_data_phase_ready_delay* to set the delay of the BREADY and RREADY signals.

The following sections describe the main processes and variables:

m_wr_resp_phase_ready_delay

The *m_wr_resp_phase_ready_delay* variable holds the BREADY signal delay. The delay value extends the length of the write response phase by a number of ACLK cycles.

Example 11-7 below shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.

Example 11-7. m_wr_resp_phase_ready_delay

```
-- Variable : m_wr_resp_phase_ready_delay
signal m_wr_resp_phase_ready_delay :integer := 2;
```

m_rd_data_phase_ready_delay

The *m_rd_data_phase_ready_delay* variable holds the RREADY signal delay. The delay value extends the length of each read data phase (beat) in a read data burst by a number of ACLK cycles.

Example 11-8 below shows the RREADY signal delayed by two ACLK cycles. You can edit this variable to change the RREADY signal delay.

Example 11-8. m_rd_data_phase_ready_delay

```
-- Variable : m_rd_data_phase_ready_delay
signal m_rd_data_phase_ready_delay : integer := 2;
```

Configuration and Initialization

The master test process creates and executes read and write transactions. The whole process runs concurrently with other processes in the test program, using the *path_id = AXI4_PATH_0*. See “Overloaded Procedure Common Arguments” on page 221 for details of *path_id*.

The process waits for the ARESETn signal to be deasserted, followed by a positive ACLK edge, as shown in Example 11-10. This satisfies the protocol requirements in Section A3.1.2 of the AXI Protocol Specification.

Example 11-9. Configuration and Initialization

```
-- Master test
process
   variable tr_id: integer;
   variable data_words :  std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
   variable lp: line;
   begin
      wait_on(AXI4_RESET_0_TO_1, index, axi4_tr_if_0(index));
      wait_on(AXI4_CLOCK_POSEDGE, index, axi4_tr_if_0(index));
```

Create and Execute Write Transactions

To generate AXI4 protocol traffic, the master test program must create a transaction before executing it. The code shown in Example 11-10 calls the *create_write_transaction()* procedure, providing only the start address argument of the transaction. The optional burst-length argument automatically defaults to a value of zero, indicating a burst length of a single beat.

This example has an AXI4 write data bus width of 32 bits; therefore, a single beat of data conveys 4 bytes across the data bus. The call to the *set_data_words()* procedure sets the first element of the *data_words[0]* transaction field with the value 1 on byte lane 1, with a result of *x"0000_0100"*. However, the AXI4 protocol permits narrow transfers with the use of the write strobes signal WSTRB to indicate which byte lane contains valid write data, and therefore indicates to the slave DUT which data byte lane will be written into memory. The write strobes WSTRB signal indicates to the slave which byte lane contains valid write data to be written to the slave memory. Similarly, you can call the *set_write_strobes()* procedure to set the first element of the *write_strobes* transaction field with the value 2, indicating that only byte lane 1 contains valid data. Calling the *execute_transaction()* procedure executes the transaction on the protocol signals.

All other transaction fields default to legal protocol values (see *create_write_transaction()* procedure for details).

Example 11-10. Create and Execute Write Transactions

```
-- 4 x Writes
-- Write data value 1 on byte lanes 1 to address 1.
create_write_transaction(1, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"00000100";
set_data_words(data_words, tr_id, index, axi4_tr_if_0(index));
set_write_strobes(2, tr_id, index, axi4_tr_if_0(index));
report "master_test_program: Writing data (1) to address (1)";

-- By default it will run in Blocking mode
execute_transaction(tr_id, index, axi4_tr_if_0(index));
```

In the complete master test program, three subsequent write transactions are created and executed in a similar manner to Example 11-10.

Create and Execute Read Transactions

The code excerpt in Example 11-11 reads the data that has been previously written into the slave memory. The master test program first creates a read transaction by calling the *create_read_transaction()* procedure, providing only the start address argument. The optional burst length automatically defaults to a value of zero, indicating a burst length of a single beat.

The *set_size()* procedure is then called to set the transaction *size* field to a single byte (AXI4_BYTES_1), and the *set_id()* procedure call sets the transaction *id* field to be 1. The read transaction is then executed on the protocol signals by calling the *execute_transaction()* procedure.

The read data is obtained using the *get_data_words()* procedure to get the *data_words* transaction field value. The result of the read data is compared with the expected data, and a message displays the transcript.

Example 11-11. Create and Execute Read Transactions

```
--4 x Reads
--Read data from address 1.
create_read_transaction(1, tr_id, index, axi4_tr_if_0(index));
set_id(1, tr_id, index, axi4_tr_if_0(index));
set_size(AXI4_BYTES_1, tr_id, index, axi4_tr_if_0(index));
execute_transaction(tr_id, index, axi4_tr_if_0(index));

get_data_words(data_words, tr_id, index, axi4_tr_if_0(index));
if(data_words(31 downto 0) = x"00000100") then
   report "master_test_program: Read correct data (1) at address (1)";
else
   hwrite(lp, data_words(31 downto 0));
   report "master_test_program: Error: Expected data (1) at address 1, but
got " & lp.all;
end if;
```

In the complete master test program, three subsequent read transactions are created and executed in a similar manner to Example 11-11.

Create and Execute Write Burst Transactions

The code excerpt in Example 11-12 calls the *create_write_transaction()* procedure to create a write burst transaction by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 7+1=8.

Note

The burst length argument passed to the *create_write_transaction()* procedure is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the AWLEN protocol signals.

The *set_data_words()* procedure is then called eight times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst, therefore a *for .. loop* calls the *set_write_strobes()* procedure to set the *write_strobes* fields of the transaction to 15 for each beat of the burst.

The write transaction is then executed onto the protocol signals.

Example 11-12. Create and Execute Write Burst Transactions

```
--  Write data burst length of 7 to start address 16.
create_write_transaction(16, 7, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE0ACE1";
set_data_words(data_words, 0, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE2ACE3";
set_data_words(data_words, 1, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE4ACE5";
set_data_words(data_words, 2, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE6ACE7";
set_data_words(data_words, 3, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE8ACE9";
set_data_words(data_words, 4, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACEAACEB";
set_data_words(data_words, 5, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACECACED";
set_data_words(data_words, 6, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACEEACEF";
set_data_words(data_words, 7, tr_id, index, axi4_tr_if_0(index));
for i in  0 to 7 loop
   set_write_strobes(15, i, tr_id, index, axi4_tr_if_0(index));
end loop;
execute_transaction(tr_id, index, axi4_tr_if_0(index));
```

In the complete master test program, a subsequent write data burst transaction with a start address of 128 is created and executed in a similar manner to Example 11-12.

Create and Execute Read Burst Transactions

The code excerpt in Example 11-13 reads the first two data beats from the data burst that has been previously written into the slave memory. The call to the *create_read_transaction()* procedure creates the read burst transaction by providing the start address and burst length arguments. The actual length of the burst on the protocol signals is 1+1=2.

Note

The burst length argument passed to the *create_read_transaction()* procedure is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the ARLEN protocol signals.

The read transaction is then executed onto the protocol signals by calling the *execute_transaction()* procedure. The read data is obtained by calling the *get_data_words()*

procedure twice to get the *data_words* transaction field values. The result of the read data is compared with the expected data, and a message displays the transcript.

Example 11-13. Create and Execute Read Burst Transactions

```
create_read_transaction(16, 1, tr_id, index, axi4_tr_if_0(index));
execute_transaction(tr_id, index, axi4_tr_if_0(index));

get_data_words(data_words, 0, tr_id, index, axi4_tr_if_0(index));
if(data_words(31 downto 0) = x"ACE0ACE1") then
   report "master_test_program: Read correct data (hACE0ACE1) at
           address (16)";
else
   hwrite(lp, data_words(31 downto 0));
   report "master_test_program: Error: Expected data (hACE0ACE1) at
           address (16), but got " & lp.all;
end if;
get_data_words(data_words, 1, tr_id, index, axi4_tr_if_0(index));
if(data_words(31 downto 0) = x"ACE2ACE3") then
   report "master_test_program: Read correct data (hACE2ACE3) at
           address (20)";
else
   hwrite(lp, data_words(31 downto 0));
   report "master_test_program: Error: Expected data (hACE2ACE3) at
           address (20), but got " & lp.all;
end if;
```

In the complete master test program, a subsequent read transaction with a start address of 128 is created and executed in a similar manner to Example 11-13.

Create and Execute Outstanding Write Burst Transactions

The code excerpt in Example 11-14 uses the AXI4 Master BFM *create_write_transaction()* procedure to create a write burst transaction by providing the start address and burst length arguments. The actual length of the burst on the protocol wires is 3+1=4.

Note

The burst length argument passed to the *create_read_transaction()* procedure is 1 less than the number of transfers (beats) in the burst. This aligns the burst length argument value with the value placed on the ARLEN protocol signals.

The *set_data_words()* procedure is then called four times to set the *data_words* field of the write transaction for each beat of the data burst. For this write transaction, all data byte lanes contain valid data on each beat of the data burst, therefore a *for* loop calls the *set_write_strobes()* procedure to set the *write_strobes* fields of the transaction to 15.

The call to the *set_operation_mode()* procedure configures the transaction to be nonblocking by setting the *operation_mode* field to AXI4_TRANSACTION_NON_BLOCKING.

The write transaction is then executed onto the protocol signals by calling the *execute_transaction()* procedure. The executed transaction will be nonblocking allowing subsequent address phase transactions to be executed before the current write data burst has completed. This allows outstanding write transaction stimulus to be created.

Example 11-14. Create and Execute Outstanding Write Burst Transactions

```
create_write_transaction(0, 3, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE0ACE1";
set_data_words(data_words, 0, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE2ACE3";
set_data_words(data_words, 1, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE4ACE5";
set_data_words(data_words, 2, tr_id, index, axi4_tr_if_0(index));
data_words(31 downto 0) := x"ACE6ACE7";
set_data_words(data_words, 3, tr_id, index, axi4_tr_if_0(index));
for i in  0 to 3 loop
   set_write_strobes(15, i, tr_id, index, axi4_tr_if_0(index));
end loop;
set_operation_mode(AXI4_TRANSACTION_NON_BLOCKING, tr_id, index,
                        axi4_tr_if_0(index));
execute_transaction(tr_id, index, axi4_tr_if_0(index));
```

In the complete master test program, subsequent write transactions are created and executed in a similar manner to Example 11-4.

handle_write_resp_ready

The *handle write response ready* process handles the BREADY signal for the write response channel. The whole process runs concurrently with other processes in the test program, using the *path_id = AXI4_PATH_5*. See “Overloaded Procedure Common Arguments” on page 221 for details of *path_id*, as shown in the Example 11-15.

The initial wait for the ARESETn signal to be deactivated, followed by a positive ACLK edge, satisfies the protocol requirement detailed in Section A3.1.2 of the AXI Protocol Specification.

The BREADY signal is deasserted using the nonblocking call to the *execute_write_resp_ready()* procedure and waits for a write channel response phase to occur with a call to the blocking *get_write_response_cycle()* procedure. A received write response phase indicates that the BVALID signal has been asserted, triggering the starting point for the delay of the BREADY signal. In a *loop* it delays the assertion of BREADY based on the setting of the *m_wr_resp_phase_ready_delay* variable. After the delay, another call to the *execute_write_resp_ready()* procedure to assert the BREADY signal completes the BREADY handling.

Example 11-15. Process handle_write_resp_ready

```
-- handle_write_resp_ready : write response ready through path 5.
-- This method assert/de-assert the write response channel ready signal.
-- Assertion and de-assertion is done based on following variable's value:
-- m_wr_resp_phase_ready_delay
process
   variable tmp_ready_delay : integer;
begin
   wait_on(AXI4_RESET_0_TO_1, index, AXI4_PATH_5, axi4_tr_if_5(index));
   wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5, axi4_tr_if_5(index));
   loop
      wait until m_wr_resp_phase_ready_delay > 0;
      tmp_ready_delay := m_wr_resp_phase_ready_delay;
      execute_write_resp_ready(0, 1, index, AXI4_PATH_5,
axi4_tr_if_5(index));
      get_write_response_cycle(index, AXI4_PATH_5, axi4_tr_if_5(index));
      if(tmp_ready_delay > 1) then
         for i in  0 to tmp_ready_delay-2 loop
            wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5,
axi4_tr_if_5(index));
         end loop;
      end if;
      execute_write_resp_ready(1, 1, index, AXI4_PATH_5,
axi4_tr_if_5(index));
   end loop;
   wait;
end process;
```

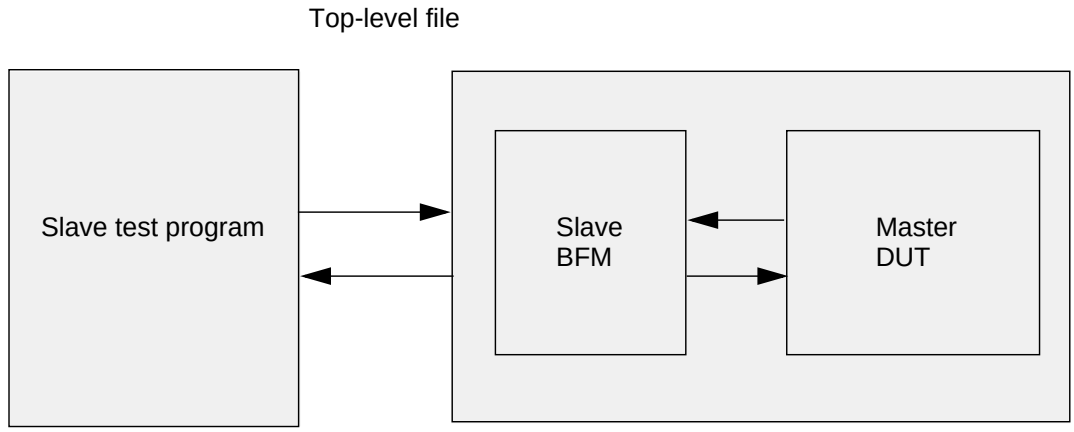
handle_read_data_ready

The *handle read data ready* process handles the RREADY signal for the read data channel. It delays the assertion of the RREADY signal based on the setting of the *m_rd_data_phase_ready_delay* variable. The whole process runs concurrently with other processes in the test program, using the *path_id = AXI4_PATH_6*. See “Overloaded Procedure Common Arguments” on page 221 for details of *path_id*, and is similar in operation to the *handle_write_resp_ready* procedure.

Verifying a Master DUT

A master DUT component is connected to a slave BFM at the signal-level. A slave test program, written at the transaction-level, generates stimulus via the slave BFM to verify the master DUT. Figure 11-2 illustrates a typical top-level test bench environment.

Figure 11-2. Master DUT Top-Level Test Bench Environment



In this example, the slave test program is a simple memory model.

A top-level file instantiates and connects all the components required to test and monitor the DUT, and controls the system clock (ACLK) and reset (ARESETn) signals.

AXI3 BFM Slave Test Program

The slave test program is a memory model and contains two APIs:

- AXI3 Basic Slave API Definition

  The *AXI3 Basic Slave API Definition* allows you to create a wide range of stimulus scenarios to test a master DUT. This simple API design illustrates the creation of slave stimulus based on the default response of OKAY to master read and write transactions.

- AXI3 Advanced Slave API Definition
- The *AXI3 Advanced Slave API Definition* allows you to create additional response scenarios to transactions. For example, a successful exclusive transaction requires an EXOKAY response.

Note

For a complete code example of this slave test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi3\qsys-examples\ex1_back_to_back_vhd\slave_test_program.vhd

AXI3 Basic Slave API Definition

The Slave Test Program Basic API contains the following:

- Procedures that read and write a byte of data to *internal memory do_byte_read()* and *do_byte_write()*, respectively.
- Procedures to configure the AXI3 protocol channel handshake delays *set_read_address_ready_delay()*, set_write_address_ready_delay(), *set_write_data_ready_delay()*, *set_read_data_valid_delay()* and *set_wr_resp_valid_delay()*.
- Procedures to process read and write transactions, *process_read* and *process write*, respectively. If you need to create other responses, such as EXOKAY, DECERR, or SLVERR, then you must edit these procedures to provide the required response.
- A *slave_mode* transaction field to control the behavior of reading and writing to the internal memory.

The internal memory for the slave is defined as an array of 8 bits, so that each byte of data is stored as an address/data pair.

Example 11-16. internal memory

```
type memory_t is array (0 to 2**16-1) of std_logic_vector(7 downto 0);
shared variable mem : memory_t;
```

The *do_byte_read()* procedure, when called, reads a *data* byte from the internal memory *mem*, given an address location *addr*, as demonstrated in Example 11-17.

You can edit this procedure to modify the way the read data is extracted from the internal memory.

Example 11-17. do_byte_read()

```
-- Procedure : do_byte_read
-- Procedure to provide read data byte from memory at particular input
-- address
procedure do_byte_read
(
      addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
      data : out std_logic_vector(7 downto 0)
) is
begin
      data := mem(to_integer(addr));
end do_byte_read;
```

The *do_byte_write()* procedure, when called, writes a *data* byte to the internal memory *mem*, given an address location *addr*, as Example 11-18 illustrates.

You can edit this procedure to modify the way the write data is stored in the internal memory.

Example 11-18. do_byte_write()

```
-- Procedure : do_byte_write
-- Procedure to write data byte to memory at particular input address
procedure do_byte_write
(
      addr : in std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
      data : in std_logic_vector(7 downto 0)
) is
begin
      mem(to_integer(addr)) := data;
end do_byte_write;
```

The *set_read_address_ready_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called it configures the ARREADY handshake signal to be delayed by a number of ACLK cycles that extends the length of the read address phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field (refer to "*AXI3 BFM Delay Mode*" on page 49 for details). Example 11-19 demonstrates setting the ARREADY signal delay by four ACLK cycles.

You can edit this procedure to change the ARREADY signal delay.

Example 11-19. set_read_address_ready_delay()

```
-- Procedure : set_read_address_ready_delay
-- This is used to set read address phase ready delay to extend phase
procedure set_read_address_ready_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(4, id, index, tr_if);
end set_read_address_ready_delay;

procedure set_read_address_ready_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(4, id, index, path_id, tr_if);
end set_read_address_ready_delay;
```

The *set_write_address_ready_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called, it configures the AWREADY handshake signal to be delayed by a number of ACLK cycles, which extends the length of the write address phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details. Example 11-20 demonstrates setting the AWREADY signal delay by two ACLK cycles.

You can edit this procedure to change the AWREADY signal delay.

Example 11-20. set_write_address_ready_delay()

```
-- Procedure : set_write_address_ready_delay
-- This is used to set write address phase ready delay to extend phase
procedure set_write_address_ready_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(2, id, index, tr_if);
end set_write_address_ready_delay;

procedure set_write_address_ready_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
begin
      set_address_ready_delay(2, id, index, path_id, tr_if);
end set_write_address_ready_delay;
```

The *set_write_data_ready_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called, it configures the WREADY signal handshake to be delayed by a number of ACLK cycles, which extends the length of each write data phase (beat) in a write data burst. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details.

For each write data phase (beat), the delay value of the WREADY signal is stored in an element of the *data_ready_delay[]* array for the transaction, as demonstrated in Example 11-21.

You can edit this procedure to change the WREADY signal delays.

Example 11-21. set_write_data_ready_delay()

```
-- Procedure : set_write_data_ready_delay
-- This will set the ready delays for each write data phase in a write data
-- burst
procedure set_write_data_ready_delay
(
      id : integer;
      signal tr_if : inout axi_vhd_if_struct_t
) is
      variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, tr_if);
    for i in  0 to burst_length loop
      set_data_ready_delay(i, i, id, index, tr_if);
    end loop;
end set_write_data_ready_delay;
```

```
procedure set_write_data_ready_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
    variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, path_id, tr_if);
    for i in  0 to burst_length loop
      set_data_ready_delay(i, i, id, index, path_id, tr_if);
    end loop;
end set_write_data_ready_delay;
```

The set_read_data_valid_delay() procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called, it configures the RVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of each read data phase (beat) in a read data burst. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 or details.

For each read data phase (beat), the delay value of the RVALID signal is stored in an element of the *data_valid_delay[]* array for the transaction, as demonstrated in Example 11-22.

You can edit this procedure to change the RVALID signals delays.

Example 11-22. set_read_data_valid_delay()

```
-- Procedure : set_read_data_valid_delay
-- This will set the ready delays for each write data phase in a write data
-- burst
procedure set_read_data_valid_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
    variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, tr_if);
    for i in  0 to burst_length loop
      set_data_valid_delay(i, i, id, index, tr_if);
    end loop;
end set_read_data_valid_delay;

procedure set_read_data_valid_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
variable burst_length : integer;
begin
    get_burst_length(burst_length, id, index, path_id, tr_if);
    for i in  0 to burst_length loop
      set_data_valid_delay(i, i, id, index, path_id, tr_if);
    end loop;
end set_read_data_valid_delay;
```

The *set_wr_resp_valid_delay()* procedure has two prototypes, one for multiple process threads by providing the *path_id* argument. When called, it configures the BREADY signal handshake to be delayed by a number of ACLK cycles, which extends the length of the write response phase. The starting point of the delay is determined by the configuration of the *delay_mode* operational transaction field. Refer to "AXI3 BFM Delay Mode" on page 49 for details. Example 11-23 demonstrates setting the BREADY signal delay by two ACLK cycles.

You can edit this procedure to change the BREADY signal delay.

Example 11-23. set_wr_resp_valid_delay()

```
-- Procedure : set_wr_resp_valid_delay
-- This is used to set write response phase valid delay to start driving
-- write response phase after specified delay.
procedure set_wr_resp_valid_delay
(
      id : integer; signal tr_if : inout axi_vhd_if_struct_t
) is
begin
    set_write_response_valid_delay(0, id, index, tr_if);
end set_wr_resp_valid_delay;

procedure set_wr_resp_valid_delay
(
      id : integer; path_id : in axi_path_t;
      signal tr_if : inout axi_vhd_if_struct_t
) is
begin
    set_write_response_valid_delay(0, id, index, path_id, tr_if);
end set_wr_resp_valid_delay;
```

There is a *slave_mode* transaction field that you can configure to control the behavior of reading and writing to the *internal memory*. It has two modes: AXI_TRANSACTION_SLAVE and AXI_PHASE_SLAVE, as shown in Example 11-24.

Example 11-24. slave_mode

```
-- Slave mode type definition
type axi_slave_mode_e is (AXI_TRANSACTION_SLAVE, AXI_PHASE_SLAVE);

-- Slave mode selection : Default is transaction-level slave
signal slave_mode : axi_slave_mode_e := AXI_TRANSACTION_SLAVE;
```

The default AXI_TRANSACTION_SLAVE mode "saves up" an entire data burst and modifies the internal memory of the slave test program in zero time for the whole burst. Therefore, a read from internal memory is buffered at the beginning of the read burst for the whole burst. The buffered read data is then transmitted over the protocol signals to the master on a phase-by-phase (beat-by-beat) basis. For a write, the write data burst is buffered on a phase-by-phase (beat-by-beat) basis for the whole burst. Only at the end of the write burst are the buffered contents written to the internal memory.

The AXI_PHASE_SLAVE mode changes the slave test program internal memory on each data phase (beat). Therefore, a read from the internal memory occurs only when the read data phase (beat) actually starts on the protocol signals. For a write, data is written to the internal memory as soon as each individual write data phase (beat) completes.

Note

In addition to the above procedures, you can configure other aspects of the AXI3 slave BFM by using the procedures: "set_config()" on page 375 and "get_config()" on page 378.

Using the AXI3 Basic Slave Test Program API

As described in "AXI3 Basic Slave API Definition" on page 645, there is a set of procedures that you use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing. However, consider the following configurations when using the slave test program.

- *slave_mode* – The read and write channel interaction can cause simultaneous read and write transactions to occur at the same address. With the default *slave_mode* setting the read transaction, the data burst is buffered at the start of the burst, and the write data burst is buffered at the end of the burst. This can result in the read data being stale at the time it is transmitted over the protocol signals. If this is an undesirable result, then set the *slave_mode* to be AXI_PHASE_SLAVE.
- *delay_mode* – By default, the handshake *READY signal always follows, or is simultaneous with the *VALID signal. By configuring the *delay_mode* to be AXI_TRANS2READY, *READY before *VALID scenarios can be achieved.

AXI3 Advanced Slave API Definition

Note

You are not required to edit the following Advance Slave API unless you require a different response than the default (OKAY) response.

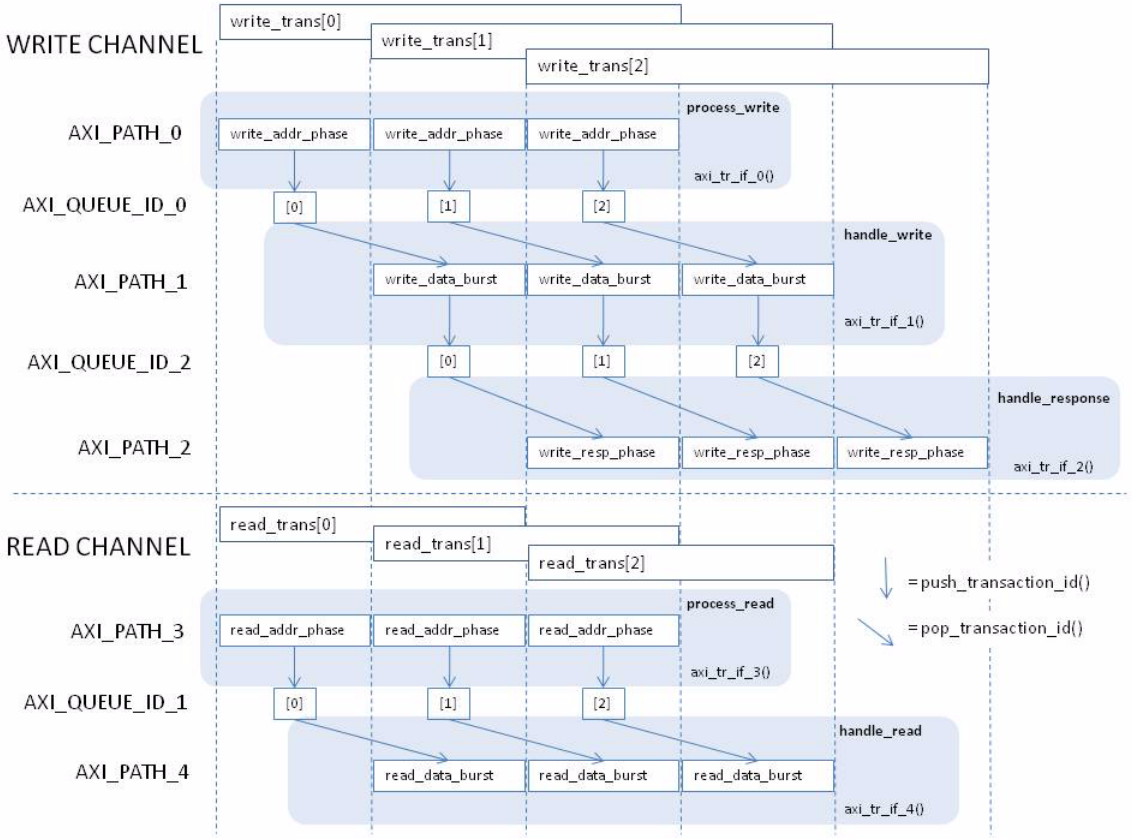
The remaining section of this tutorial presents a walk-through of the Advanced Slave API. It consists of three processes for write transactions and two for read transactions. You are not required to edit the following Advance Slave API, unless you require a different response than the default (OKAY) response.

The Advanced Slave API is capable of handling pipelined transactions. Pipelining can occur when a transaction starts before a previous transaction has completed. Therefore, a write transaction that starts before a previous write transaction has completed can be pipelined. Figure 11-3 shows the write channel having three concurrent *write_trans* transactions, with the

write_addr_phase[2], *write_data_burst[1]* and *write_response_phase[0]* being concurrently active on the write address, data and response channels, respectively.

Similarly, a read transaction that starts before a previous read transaction has completed can be pipelined. Figure 11-3 shows the read channel having two concurrent *read_trans* transactions, whereby the *read_addr_phase[1]* and *read_data_burst[0]* are concurrently active on the read address and data channels, respectively.

Figure 11-3. Slave Test Program Advanced API Tasks



The *process write* code extract demonstrates how to create and process multiple write transactions. Each write transaction has a unique *transaction_id* number associated with it that increments for each new transaction. A *write_trans* variable is defined to hold this *transaction_id*.

The processing of write transactions initially configures the maximum number of outstanding write transactions that can exist. An ACLK period after the ARESETn signal is inactive starts a loop to create a slave transaction for the BFM indexed by the *index* argument. In the loop, the delay for the AWREADY signal is set by the *set_write_address_ready_delay* procedure and

then it waits for a write address phase to occur using the *get_write_addr_phase* procedure. The transaction record is then pushed onto a transaction queue with its unique *write_trans* transaction index and queue identifier AXI_QUEUE_ID_0 by the *push_transaction_id* procedure.

The loop then completes and starts again by creating a new transaction and waiting for another write address phase to occur, as shown in Example 11-25.

Example 11-25. process write

```
-- process_write : write address phase through path 0
-- This process keep receiving write address phases and pushes
-- the transaction into a queue via the push_transaction_id procedure.
process
    variable write_trans : integer;
begin
   wait_on(AXI_RESET_0_TO_1, index, axi_tr_if_0(index));
   wait_on(AXI_CLOCK_POSEDGE, index, axi_tr_if_0(index));
   loop
      create_slave_transaction(write_trans, index, axi_tr_if_0(index));
      set_write_address_ready_delay(write_trans, axi_tr_if_0(index));
      get_write_addr_phase(write_trans, index, axi_tr_if_0(index));
      push_transaction_id(write_trans, AXI_QUEUE_ID_0, index,
axi_tr_if_0(index));
    end loop;
    wait;
end process;
```

The *handle_write* code extract demonstrates how to write a data burst to an internal memory using buffered and unbuffered approaches.

To perform pipelining of the AXI3 address and data phases of one transaction concurrently with another transaction, the Advanced Slave API provides procedures with an optional *path_id* argument. This permits multiple threads of code execution in the Slave API. The following *handle_write* code uses *path_id* = *AXI_PATH_1,* as an example.

Initially, a number of local variables is defined to hold the transaction index *write_trans* and some of the transaction fields, such as burst length and data. In a loop, the existing record of the *write_trans* transaction is popped from the queue identifier AXI_QUEUE_ID_0 via the *pop_transaction_id* procedure. The delay for the WREADY signal is then set by the *set_write_data_ready_delay* procedure.

If the *slave_mode* is configured to AXI_TRANSACTION_SLAVE (buffered), the code waits for a complete write data burst by the *get_write_data_burst* procedure before continuing. The burst length of the write data burst is obtained using the *get_burst_length* procedure. The resulting *burst_length* is then used to set the subsequent maximum inner loop *i* count for the number of data beats in the burst.

Loop *i* gets the address and data pairs from the transaction via the *get_write_addr_data* procedure before calling the *do_byte_write* procedure, which writes the *data* byte into the memory *mem* at the corresponding *addr* address. If the number of bytes to be written (for this beat) is more than one, then loop *j* writes the remaining bytes of this beat into the memory *mem*. Loop *i* then repeats for each data beat up to the length of the data burst.

If the *slave_mode* is configured to AXI_PHASE_SLAVE (unbuffered), the code waits for a single write data phase (beat) to complete via the *get_write_data_phase* procedure, in a *while* loop. The address and data pairs from the transaction are obtained by the *get_write_addr_data* procedure before calling the *do_byte_write* procedure, which writes the *data* byte into the memory *mem* at the corresponding *addr* address. If the number of bytes to be written (for this beat) is more than one, then loop *j* writes the remaining bytes of this beat into the memory *mem*. The *while* loop then repeats, waiting for another data phase (beat), if this is not the last data phase (beat) in the burst.

The transaction record, so far, is then pushed onto a new transaction queue identifier AXI_QUEUE_ID_2 by the *push_transaction_id* procedure, which is then ready for the slave to execute a response back to the master (refer to Example 11-26).

Example 11-26. handle_write

```
-- handle_write : write data phase through path 1
-- This process receives write data burst, or write phases (beats).
-- The slave_mode configuration controls when the write data is passed to
-- memory.
  process
    variable write_trans: integer;
    variable byte_length : integer;
    variable burst_length : integer;
    variable addr : std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
    variable data : std_logic_vector(7 downto 0);
    variable last : integer := 0;
    variable loop_i : integer := 0;
  begin
    loop
      pop_transaction_id(write_trans, AXI_QUEUE_ID_0, index, AXI_PATH_1,
axi_tr_if_1(index));
      set_write_data_ready_delay(write_trans, AXI_PATH_1,
axi_tr_if_1(index));

      if (slave_mode = AXI_TRANSACTION_SLAVE) then
        get_write_data_burst(write_trans, index, AXI_PATH_1,
axi_tr_if_1(index));
        get_burst_length(burst_length, write_trans, index, AXI_PATH_1,
axi_tr_if_1(index));
        for i in  0 to burst_length loop
         get_write_addr_data(write_trans,  i, 0, byte_length, addr, data,
index, AXI_PATH_1, axi_tr_if_1(index));
          do_byte_write(addr, data);
          if byte_length > 1 then
            for j in 1 to byte_length-1 loop
               get_write_addr_data(write_trans,  i, j, byte_length, addr,
data, index, AXI_PATH_1, axi_tr_if_1(index));
               do_byte_write(addr, data);
            end loop;
          end if;
        end loop;
      else
        last := 0;
        loop_i := 0;
        while(last = 0) loop
          get_write_data_phase(write_trans, loop_i, last, index,
AXI_PATH_1, axi_tr_if_1(index));
get_write_addr_data(write_trans,  loop_i, 0, byte_length, addr, data,
index, AXI_PATH_1, axi_tr_if_1(index));
          do_byte_write(addr, data);
          if byte_length > 1 then
            for j in 1 to byte_length-1 loop
               get_write_addr_data(write_trans,  loop_i, j, byte_length,
addr, data, index, AXI_PATH_1, axi_tr_if_1(index));
               do_byte_write(addr, data);
            end loop;
          end if;
            loop_i := loop_i + 1;
        end loop;
      end if;
```

```
      push_transaction_id(write_trans, AXI_QUEUE_ID_2, index, AXI_PATH_1,
axi_tr_if_1(index));
    end loop;
    wait;
  end process;
```

The *handle_response* code extract demonstrates how to respond to a master write transaction using a different *path_id = AXI_PATH_2* than the *path_id* used for the address and data phases of the same transaction. This allows the slave to execute a write transaction response in a different order than the order received for a particular *write_trans* index number.

A *write_trans* variable is defined to hold this transaction index number before entering a loop to pop a write transaction from AXI_QUEUE_ID_2 via the *pop_transaction_id* procedure call. The delay for the BVALID signal is then set via the *set_wr_resp_valid_delay* procedure. The response phase is then executed by the *execute_write_response_phase* procedure call, as shown in Example 11-27.

Example 11-27. handle_response

```
-- handle_response : write response phase through path 2
-- This method sends the write response phase
process
    variable write_trans: integer;
  begin
    loop
      pop_transaction_id(write_trans, AXI_QUEUE_ID_2, index, AXI_PATH_2,
axi_tr_if_2(index));
      set_wr_resp_valid_delay(write_trans, AXI_PATH_2,
axi_tr_if_2(index));
      execute_write_response_phase(write_trans, index, AXI_PATH_2,
axi_tr_if_2(index));
    end loop;
    wait;
  end process;
```

The processing of read transactions works in a similar way as that described above for write transactions. There are two processes *process_read* and *handle read*.

The main difference between write and read transaction handling is that the read transaction retrieves the read data burst from the internal memory *mem* at the start of the data burst, or on a phase-by-phase (beat-by-beat) basis, depending on the *slave_mode* configuration setting. In addition, AXI3 has a read response per read data phase (beat); so unlike the write, read does not require a separate read response handling process. Refer to Example 11-28 and Example 11-29.

Example 11-28. process_read

```
-- process_read : read address phase through path 3
  -- This process keep receiving read address phase and push the
transaction into queue through
  -- push_transaction_id API.
  process
    variable read_trans: integer;
  begin
   wait_on(AXI_RESET_0_TO_1, index, AXI_PATH_3, axi_tr_if_3(index));
   wait_on(AXI_CLOCK_POSEDGE, index, AXI_PATH_3, axi_tr_if_3(index));
      loop
      create_slave_transaction(read_trans, index, AXI_PATH_3,
axi_tr_if_3(index));
      set_read_address_ready_delay(read_trans, AXI_PATH_3,
axi_tr_if_3(index));
      get_read_addr_phase(read_trans, index, AXI_PATH_3,
axi_tr_if_3(index));
      push_transaction_id(read_trans, AXI_QUEUE_ID_1, index, AXI_PATH_3,
axi_tr_if_3(index));
    end loop;
    wait;
  end process;
```

Example 11-29. handle read

```
-- handle_read : read data and response through path 4
  -- This process reads data from memory and send read data/response
either at
  -- burst or phase level depending upon slave working mode.
  process
    variable read_trans: integer;
    variable burst_length : integer;
    variable byte_length : integer;
    variable addr : std_logic_vector(AXI_MAX_BIT_SIZE-1 downto 0);
    variable data : std_logic_vector(7 downto 0);
  begin
    loop
      pop_transaction_id(read_trans, AXI_QUEUE_ID_1, index, AXI_PATH_4,
axi_tr_if_4(index));
      set_read_data_valid_delay(read_trans, AXI_PATH_4,
axi_tr_if_4(index));
      get_burst_length(burst_length, read_trans, index, AXI_PATH_4,
axi_tr_if_4(index));
      for i in  0 to burst_length loop
        get_read_addr(read_trans,  i, 0, byte_length, addr, index,
AXI_PATH_4, axi_tr_if_4(index));
        do_byte_read(addr, data);
        set_read_data(read_trans, i, 0, byte_length, addr, data, index,
AXI_PATH_4, axi_tr_if_4(index));
         if byte_length > 1 then
          for j in 1 to byte_length-1 loop
             get_read_addr(read_trans,  i, j, byte_length, addr, index,
AXI_PATH_4, axi_tr_if_4(index));
             do_byte_read(addr, data);
             set_read_data(read_trans, i, j, byte_length, addr, data,
index, AXI_PATH_4, axi_tr_if_4(index));
            end loop;
        end if;
         if slave_mode = AXI_PHASE_SLAVE then
            execute_read_data_phase(read_trans, i, index, AXI_PATH_4,
axi_tr_if_4(index));
         end if;
      end loop;
      if slave_mode = AXI_TRANSACTION_SLAVE then
        execute_read_data_burst(read_trans, index, AXI_PATH_4,
axi_tr_if_4(index));
      end if;
    end loop;
    wait;
  end process;
```

AXI4 BFM Slave Test Program

The slave test program is a memory model that contains two APIs:

- AXI4 Basic Slave API Definition

The AXI4 Basic Slave API Definition allows you to create a wide range of stimulus scenarios to test a master DUT. This API definition simplifies the creation of slave stimulus based on the default response of OKAY to master read and write transactions.

- AXI4 Advanced Slave API Definition.

  The AXI4 Advanced Slave API Definition allows you to create additional response scenarios to transactions. For example, a successful exclusive transaction requires an EXOKAY response.

Note

For a complete code example of this slave test program, refer to the installed Mentor VIP – Intel FPGA Edition kit in the following directory:
<install_dir>\mentor_vip_ae\axi4\qsys-examples\ex1_back_to_back_vhd\slave_test_program.vhd

AXI4 Basic Slave API Definition

The Basic Slave Test Program API contains the following elements:

- Procedures *m_wr_addr_phase_ready_delay* and *do_byte_write()* that read and write a byte of data to Internal Memory, respectively.
- Procedures *set_read_data_valid_delay()* and *set_wr_resp_valid_delay()* to configure the delay of the read data channel RVALID, and write response channel BVALID signals, respectively.
- Variables *m_wr_addr_phase_ready_delay* and *m_rd_addr_phase_ready_delay* to configure the delay of the read/write address channel AWVALID/ARVALID signals, and *m_wr_data_phase_ready_delay* to configure the delay of the write response channel BVALID signal.
- A *slave_mode* variable that configures the behavior of reading and writing to internal memory.

Internal Memory

The internal memory for the slave is defined as an array of 8 bits, so that each byte of data is stored as an address/data pair.

Example 11-30. Internal Memory

```
type memory_t is array (0 to 2**16-1) of std_logic_vector(7 downto 0);
shared variable mem : memory_t;
```

do_byte_read()

The *do_byte_read()* procedure reads a *data* byte from the Internal Memory *mem* given an address location *addr*, as shown below.

You can edit this procedure to modify the way the read data is extracted from the internal memory.

```
-- Procedure : do_byte_read
-- Procedure to provide read data byte from memory at particular input
-- address
procedure do_byte_read(addr : in std_logic_vector(AXI4_MAX_BIT_SIZE-1
downto 0); data : out std_logic_vector(7 downto 0)) is
begin
   data := mem(to_integer(addr));
end do_byte_read;
```

do_byte_write()

The *do_byte_write()* procedure when called writes a *data* byte to the Internal Memory *mem* given an address location *addr*, as shown below.

You can edit this procedure to modify the way the write data is stored in the internal memory.

```
-- Procedure : do_byte_write
-- Procedure to write data byte to memory at particular input address
procedure do_byte_write(addr : in std_logic_vector(AXI4_MAX_BIT_SIZE-1
downto 0);  data : in std_logic_vector(7 downto 0)) is
begin
   mem(to_integer(addr)) := data;
end do_byte_write;
```

m_wr_addr_phase_ready_delay

The *m_wr_addr_phase_ready_delay* variable holds the AWREADY signal delay. The delay value extends the length of the write address phase by a number of ACLK cycles. The starting point of the delay is determined by the assertion of the AWVALID signal.

Example 11-31 shows the AWREADY signal delayed by two ACLK cycles. You can edit this variable to change the AWREADY signal delay.

Example 11-31. m_wr_addr_phase_ready_delay

```
-- Variable : m_wr_addr_phase_ready_delay
signal m_wr_addr_phase_ready_delay : integer := 2;
```

m_rd_addr_phase_ready_delay

The *m_rd_addr_phase_ready_delay* variable holds the ARREADY signal delay. The delay value extends the length of the read address phase by a number of ACLK cycles. The starting point of the delay is determined by the assertion of the ARVALID signal.

Example 11-32 shows the ARREADY signal delayed by two ACLK cycles. You can edit this variable to change the ARREADY signal delay.

Example 11-32. m_rd_addr_phase_ready_delay

```
-- Variable : m_rd_addr_phase_ready_delay
signal m_rd_addr_phase_ready_delay : integer := 2;
```

m_wr_data_phase_ready_delay

The *m_wr_data_phase_ready_delay* variable holds the WREADY signal delay. The delay value extends the length of each write data phase (beat) in a write data burst by a number of ACLK cycles. The starting point of the delay is determined by the assertion of the WVALID signal.

Example 11-33 shows the WREADY signal delayed by two ACLK cycles. You can edit this function to change the WREADY signal delay.

Example 11-33. m_wr_data_phase_ready_delay

```
-- Variable : m_wr_data_phase_ready_delay
signal m_wr_data_phase_ready_delay : integer := 2;
```

set_wr_resp_valid_delay()

The *set_wr_resp_valid_delay()* procedure has two prototypes (*path_id* is optional), and configures the BVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of the write response phase. The delay value of the BVALID signal is stored in the *write_response_valid_delay* transaction field.

Example 11-34 shows the BVALID signal delay set to two ACLK cycles. You can edit this function to change the BVALID signal delay.

Example 11-34. set_wr_resp_valid_delay()

```
-- Procedure : set_wr_resp_valid_delay
-- This is used to set write response phase valid delay to start driving
-- write response phase after specified delay.
procedure set_wr_resp_valid_delay(id : integer; path_id : in axi4_path_t;
signal tr_if : inout axi4_vhd_if_struct_t) is
begin
   set_write_response_valid_delay(2, id, index, path_id, tr_if);
end set_wr_resp_valid_delay;
```

set_read_data_valid_delay()

The *set_read_data_valid_delay()* procedure has two prototypes (*path_id* is optional), and configures the RVALID signal to be delayed by a number of ACLK cycles with the effect of delaying the start of a read data phase (beat) in a read data burst. The delay value of the RVALID signal, for each read data phase, is stored in an array element of the *data_valid_delay* transaction field.

Example 11-35 shows the RVALID signal delay incrementing by an ACLK cycle between each read data phase for the length of the burst. You can edit this function to change the RVALID signal delay for the whole read burst.

Example 11-35. set_read_data_valid_delay()

```
procedure set_read_data_valid_delay(id : integer; path_id : in
axi4_path_t; signal tr_if : inout axi4_vhd_if_struct_t) is
   variable burst_length : integer;
begin
   get_burst_length(burst_length, id, index, path_id, tr_if);
   for i in  0 to burst_length loop
      set_data_valid_delay(i, i, id, index, path_id, tr_if);
   end loop;
end set_read_data_valid_delay;
```

slave_mode

A configurable *slave_mode* signal controls the behavior of reading and writing to the Internal Memory. It has two modes—AXI4_TRANSACTION_SLAVE and AXI4_PHASE_SLAVE:

```
type axi4_slave_mode_e is (AXI4_TRANSACTION_SLAVE, AXI4_PHASE_SLAVE);

...

-- Slave mode seclection : default it is transaction level slave
signal slave_mode : axi4_slave_mode_e := AXI4_TRANSACTION_SLAVE;
```

The default AXI4_TRANSACTION_SLAVE mode “saves up” an entire data burst and modifies the slave test program internal memory in zero time for the whole burst. Therefore, a burst read from internal memory is buffered from the beginning of the burst to the end of the burst. The buffered read burst data is then transmitted over the protocol signals to the master on a phase-by-phase (beat-by-beat) basis. For a write, the data burst received over the protocol signals is buffered from the beginning of the burst to the end of the burst. At the end of the write burst, the buffered contents are written to the internal memory.

The AXI4_PHASE_SLAVE mode updates the slave test program internal memory on each data phase (beat). Therefore, a read from the internal memory occurs only when the read data phase (beat) actually starts to be transmitted on the protocol signals. For a write, data is written to the internal memory as soon as each individual write data phase (beat) is received on the protocol signals.

Note

In addition to the above variables and procedures, you can configure other aspects of the AXI4 Slave BFM by using these procedures: *“set_config()”* on page 375 and *“get_config()”* on page 378.

Using the AXI4 Basic Slave Test Program API

There is a set of variables and procedures that you can use to create stimulus scenarios based on a memory-model slave with a minimal amount of editing, as described in "AXI4 Basic Slave API Definition" on page 659.

Consider the following configuration when using the slave test program.

slave_mode – The read and write channel interaction can cause simultaneous read and write transactions to occur at the same address. With the default *slave_mode* setting, the read transaction data burst is buffered at the start of the burst, and the write data burst is buffered at the end of the burst. This can result in the read data being stale at the time it is transmitted over the protocol signals. If this is an undesirable feature, then set the *slave_mode* to be AXI4_PHASE_SLAVE.

AXI4 Advanced Slave API Definition

Note

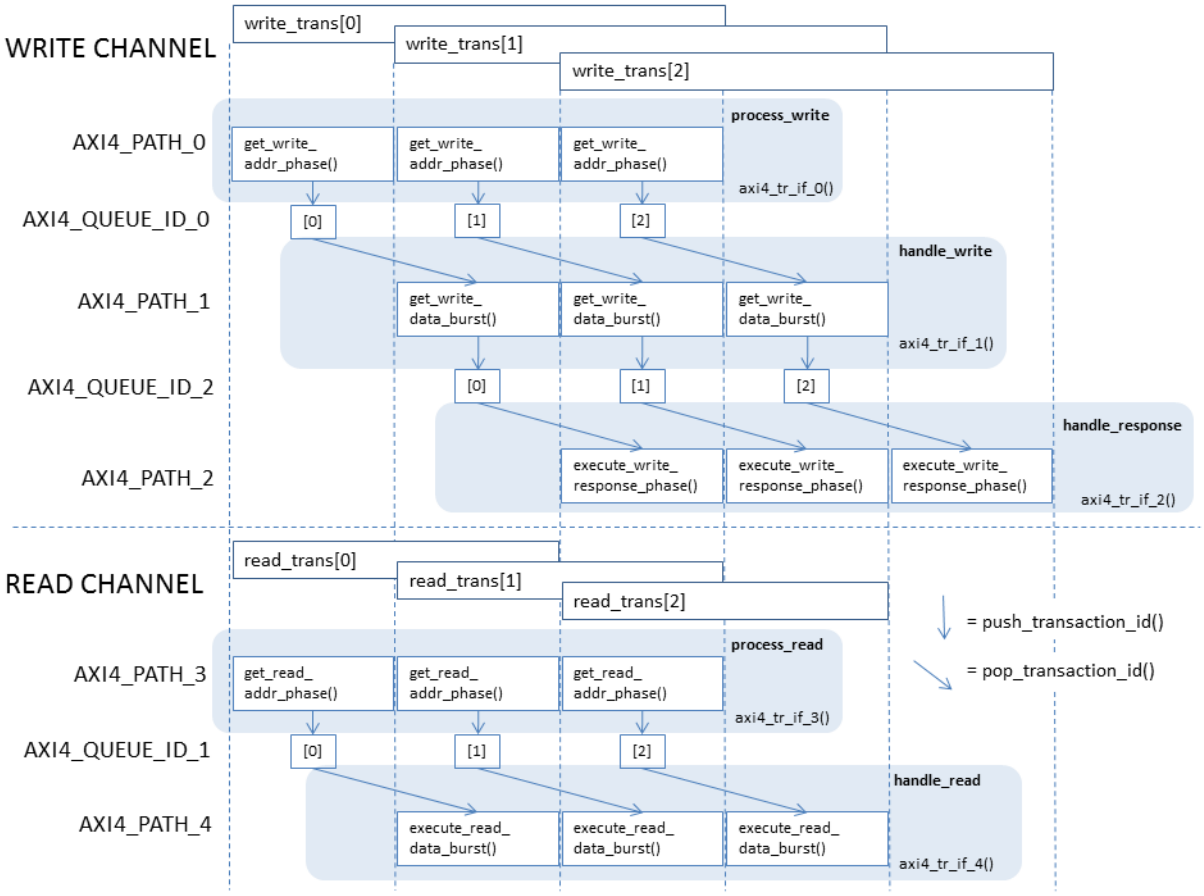
You are not required to edit the following Advance Slave API unless you require a different response than the default (OKAY) response.

The remaining section of this tutorial presents a walk-through of the Advanced Slave API in the slave test program. It consists of five main processes—*process_write*, *process_read*, *handle_write*, *handle_responsehandle_response*, and *handle_read*—in the slave test program, as shown in Figure 11-4. There are additional *handle_write_addr_ready*, *handle_read_addr_ready*, and *handle_write_data_ready* processes to handle the handshake AWREADY, ARREADY, and WREADY signals, respectively.

The Advanced Slave API is capable of handling pipelined transactions. Pipelining can occur when a transaction starts before a previous transaction has completed. Therefore, a write transaction that starts before a previous write transaction has completed can be pipelined. Figure 11-4 shows the write channel having three concurrent *write_trans* transactions, whereby the *get_write_addr_phase[2]*, *get_write_data_burst[1]* and *execute_write_response_phase[0]* are concurrently active on the write address, data and response channels, respectively.

Similarly, a read transaction that starts before a previous read transaction has completed can be pipelined. Figure 11-4 shows the read channel having two concurrent *read_trans* transactions, whereby the *get_read_addr_phase[1]* and *execute_read_data_burst[0]* are concurrently active on the read address and data channels, respectively.

Figure 11-4. Slave Test Program Advanced API Processes



process_read

The *process_read* process creates a slave transaction and receives the read address phase. It uses unique path and queue identifiers to work concurrently with other processes.

The maximum number of outstanding read transactions is configured before the processing of read transactions begins an ACLK period after the ARESETn signal is inactive, as shown in Example 11-36.

Each slave transaction has a unique *transaction_id* number associated with it that is automatically incremented for each new slave transaction created. In a *loop*, the *create_slave_transaction()* procedure call returns the *transaction_id* for the slave BFM, indexed by the *index* argument. A *read_trans* variable is previously defined to hold the *transaction_id*.

A call to the *get_read_addr_phase()* procedure blocks the code until a read address phase has completed. The call to the *push_transaction_id()* procedure pushes *read_trans* into the AXI4_QUEUE_ID_1 queue.

The *loop* completes and restarts by creating a new slave transaction and blocks for another write address phase to occur.

Example 11-36. process_read

```
-- process_read : read address phase through path 3
-- This process keep receiving read address phase and push
-- the transaction into queue through push_transaction_id API.
process
   variable read_trans: integer;
begin
   wait_on(AXI4_RESET_0_TO_1, index, AXI4_PATH_3,
            axi4_tr_if_3(index));
   wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_3,
            axi4_tr_if_3(index));
   loop
      create_slave_transaction(read_trans, index, AXI4_PATH_3,
                                    axi4_tr_if_3(index));
      get_read_addr_phase(read_trans, index, AXI4_PATH_3,
                                    axi4_tr_if_3(index));
      get_config(AXI4_CONFIG_NUM_OUTSTANDING_RD_PHASE,
                  tmp_config_num_outstanding_rd_phase, index,
                  AXI4_PATH_3, axi4_tr_if_3(index));
      push_transaction_id(read_trans, AXI4_QUEUE_ID_1, index,
                        AXI4_PATH_3, axi4_tr_if_3(index));
   end loop;
   wait;
end process;
```

handle_read

The *handle_read* process gets read data from the Internal Memory as a burst or phase (beat), depending on the *slave_mode* configuration. It uses unique path and queue identifiers to work concurrently with other processes.

In a *loop*, the *pop_transaction_id()* procedure call returns the *transaction_id* from the queue for the slave BFM, indexed by the *index* argument, as shown in Example 11-37. A *read_trans* variable is previously defined to hold the *transaction_id*. If the queue is empty, then *pop_transaction_id()* will block until content is available.

The call to *set_read_data_valid_delay()* configures the RVALID signal delay for each phase (beat) of the burst, and *get_burst_length()* returns the *burst_length* of the read transaction.

In a *loop*, the call to the *get_read_addr()* helper procedure returns the actual address *addr* for a particular byte location and the *byte_length* of the data phase (beat). This byte address is used to read the data byte from Internal Memory with the call to *do_byte_read()*, and the *set_read_data()* helper procedure sets the byte in the read transaction record. If the returned *byte_length>1*, then the code performs in the *byte_length* loop the reading and setting of the read data from internal memory for the whole of the read data phase (beat).

If the *slave_mode* configuration is set to the default of AXI4_TRANSACTION_SLAVE, then the *burst_length* loop continues until the read data has been set for the whole burst. Otherwise, the individual read data phase is executed over the protocol signals by calling *execute_read_data_phase()*.

After the *burst_length* loop is complete, *execute_read_data_burst()* is called for the default configuration of *slave_mode*, and the read burst is executed over the protocol signals.

The loop completes and restarts by waiting for another *transaction_id* to be placed into the queue.

Example 11-37. handle_read

```
-- handle_read : read data and response through path 4
-- This process reads data from memory and send read data/response either
-- at burst or phase level depending upon slave working mode.
process
   variable read_trans: integer;
   variable burst_length : integer;
   variable byte_length : integer;
   variable addr : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
   variable data : std_logic_vector(7 downto 0);
begin
   loop
      pop_transaction_id(read_trans, AXI4_QUEUE_ID_1, index, AXI4_PATH_4,
axi4_tr_if_4(index));
      set_read_data_valid_delay(read_trans, AXI4_PATH_4,
axi4_tr_if_4(index));

      get_burst_length(burst_length, read_trans, index, AXI4_PATH_4,
axi4_tr_if_4(index));
      for i in  0 to burst_length loop
         get_read_addr(read_trans,  i, 0, byte_length, addr, index,
AXI4_PATH_4, axi4_tr_if_4(index));
         do_byte_read(addr, data);
         set_read_data(read_trans, i, 0, byte_length, addr, data, index,
AXI4_PATH_4, axi4_tr_if_4(index));
         if byte_length > 1 then
            for j in 1 to byte_length-1 loop
               get_read_addr(read_trans,  i, j, byte_length, addr, index,
AXI4_PATH_4, axi4_tr_if_4(index));
               do_byte_read(addr, data);
               set_read_data(read_trans, i, j, byte_length, addr, data,
index, AXI4_PATH_4, axi4_tr_if_4(index));
            end loop;
         end if;
         if slave_mode = AXI4_PHASE_SLAVE then
            execute_read_data_phase(read_trans, i, index, AXI4_PATH_4,
axi4_tr_if_4(index));
         end if;
      end loop;
      if slave_mode = AXI4_TRANSACTION_SLAVE then
         execute_read_data_burst(read_trans, index, AXI4_PATH_4,
axi4_tr_if_4(index));
      end if;
      tmp_config_num_outstanding_rd_phase :=
                           tmp_config_num_outstanding_rd_phase - 1;
   end loop;
   wait;
end process;
```

process_write

The *process_write* process works in a similar way as previously described for *process_read*. It uses unique path and queue identifiers to work concurrently with other processes, as shown in Example 11-38.

Example 11-38. process_write

```
-- process_write : write address phase through path 0
-- This process keep receiving write address phase and push the
-- transaction into queue through push_transaction_id API.
process
   variable write_trans : integer;
begin
   wait_on(AXI4_RESET_0_TO_1, index, axi4_tr_if_0(index));
   wait_on(AXI4_CLOCK_POSEDGE, index, axi4_tr_if_0(index));
   loop
      create_slave_transaction(write_trans, index, axi4_tr_if_0(index));
      get_write_addr_phase(write_trans, index, axi4_tr_if_0(index));
      get_config(AXI4_CONFIG_NUM_OUTSTANDING_WR_PHASE,
                 tmp_config_num_outstanding_wr_phase, index,
                 AXI4_PATH_3, axi4_tr_if_0(index));
      push_transaction_id(write_trans, AXI4_QUEUE_ID_0, index,
axi4_tr_if_0(index));
   end loop;
   wait;
end process;
```

handle_write

The *handle_write* process works in a similar way to that previously described for *handle_read*. The main difference is that the write transaction handling gets the write data and stores it in the slave test program Internal Memory depending on the *slave_mode* setting, and adhering to the state of the WSTRB write strobes signals. There is an additional *pop_transaction_id()* into a queue so that the *handle_response* process can send a write response phase for the transaction, as shown in Example 11-39.

Example 11-39. handle_write

```
-- handle_write : write data phase through path 1
-- This method receive write data burst or phases for write transaction
-- depending upon slave working mode and write data to memory.
process
   variable write_trans: integer;
   variable byte_length : integer;
   variable burst_length : integer;
   variable addr : std_logic_vector(AXI4_MAX_BIT_SIZE-1 downto 0);
variable data : std_logic_vector(7 downto 0);
   variable last : integer := 0;
   variable loop_i : integer := 0;
begin
   loop
      pop_transaction_id(write_trans, AXI4_QUEUE_ID_0, index,
AXI4_PATH_1, axi4_tr_if_1(index));

      if (slave_mode = AXI4_TRANSACTION_SLAVE) then
         get_write_data_burst(write_trans, index, AXI4_PATH_1,
axi4_tr_if_1(index));
         get_burst_length(burst_length, write_trans, index, AXI4_PATH_1,
axi4_tr_if_1(index));
         for i in  0 to burst_length loop
            get_write_addr_data(write_trans,  i, 0, byte_length, addr,
data, index, AXI4_PATH_1, axi4_tr_if_1(index));
            do_byte_write(addr, data);
            if byte_length > 1 then
               for j in 1 to byte_length-1 loop
                  get_write_addr_data(write_trans,  i, j, byte_length,
addr, data, index, AXI4_PATH_1, axi4_tr_if_1(index));
                  do_byte_write(addr, data);
               end loop;
            end if;
         end loop;
      else
         last := 0;
         loop_i := 0;
         while(last = 0) loop
            get_write_data_phase(write_trans, loop_i, last, index,
AXI4_PATH_1, axi4_tr_if_1(index));
            get_write_addr_data(write_trans,  loop_i, 0, byte_length,
addr, data, index, AXI4_PATH_1, axi4_tr_if_1(index));
            do_byte_write(addr, data);
            if byte_length > 1 then
               for j in 1 to byte_length-1 loop
                  get_write_addr_data(write_trans,  loop_i, j,
byte_length, addr, data, index, AXI4_PATH_1, axi4_tr_if_1(index));
                  do_byte_write(addr, data);
               end loop;
            end if;
            loop_i := loop_i + 1;
         end loop;
      end if;
      push_transaction_id(write_trans, AXI4_QUEUE_ID_2, index,
AXI4_PATH_1, axi4_tr_if_1(index));
   end loop;
```

```
   wait;
end process;
```

handle_response

The *handle_response* process sends a response back to the master to complete a write transaction. It uses unique path and queue identifiers to work concurrently with other processes.

In a *loop*, the *pop_transaction_id()* procedure call returns the *transaction_id* from the queue for the slave BFM, indexed by the *index* argument, as shown in Example 11-40. A *write_trans* variable is previously defined to hold the *transaction_id*. If the queue is empty, then *push_transaction_id()* will block until content is available.

The call to *set_wr_resp_valid_delay()* sets the BVALID signal delay for the response prior to calling *execute_write_response_phase()* to execute the response over the protocol signals.

Example 11-40. handle_response

```
-- handle_response : write response phase through path 2
-- This method sends the write response phase
process
   variable write_trans: integer;
   begin
      loop
         pop_transaction_id(write_trans, AXI4_QUEUE_ID_2, index,
AXI4_PATH_2, axi4_tr_if_2(index));
         set_wr_resp_valid_delay(write_trans, AXI4_PATH_2,
axi4_tr_if_2(index));
         execute_write_response_phase(write_trans, index, AXI4_PATH_2,
axi4_tr_if_2(index));
          tmp_config_num_outstanding_wr_phase :=
                     tmp_config_num_outstanding_wr_phase - 1;
   end loop;
   wait;
end process;
```

handle_write_addr_ready

The *handle_write_addr_ready* process handles the AWREADY signal for the write address channel. It uses a unique path identifier to work concurrently with other processes.

The handling of the AWREADY signal begins an ACLK period after the ARESETn signal is inactive, as shown in Example 11-41. In a *loop*, the AWREADY signal is deasserted using the nonblocking call to the *execute_write_addr_ready()* procedure and blocks for a write channel address phase to occur with a call to the blocking *get_write_addr_cycle()* procedure. A received write address phase indicates that the AWVALID signal has been asserted, triggering the starting point for the delay of the AWREADY signal by the number of ACLK cycles defined by *m_wr_addr_phase_ready_delay*. Another call to the *execute_write_addr_ready()* procedure to assert the AWREADY signal completes the AWREADY handling.

Example 11-41. handle_write_addr_ready

```
-- handle_write_addr_ready : write address ready through path 5
-- This method assert/de-assert the write address channel ready signal.
-- Assertion and de-assertion is done based on m_wr_addr_phase_ready_delay
process
   variable tmp_ready_delay : integer;
   variable tmp_max_outstanding_write : integer
begin
   wait_on(AXI4_RESET_POSEDGE, index, AXI4_PATH_5, axi4_tr_if_5(index));
   wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5, axi4_tr_if_5(index));
   get_config(AXI4_CONFIG_MAX_OUTSTANDING_WR, tmp_max_outstanding_write,
              index, axi4_tr_if_5(index));
   loop
      while (tmp_config_num_outstanding_wr_phase >=
               tmp_max_outstanding_write) loop
         wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5,
                  axi4_tr_if_5(index));
      end loop;
      tmp_ready_delay := m_wr_addr_phase_ready_delay;
      execute_write_addr_ready(0, 1, index, AXI4_PATH_5,
axi4_tr_if_5(index));
      get_write_addr_cycle(index, AXI4_PATH_5, axi4_tr_if_5(index));
      if(tmp_ready_delay > 1) then
         for i in  0 to tmp_ready_delay-2 loop
            wait_on(AXI4_CLOCK_POSEDGE, index, AXI4_PATH_5,
axi4_tr_if_5(index));
         end loop;
      end if;
      execute_write_addr_ready(1, 1, index, AXI4_PATH_5,
axi4_tr_if_5(index));
   end loop;
   wait;
end process;
```

handle_read_addr_ready

The *handle_read_addr_ready* process handles the ARREADY signal for the read address channel. It uses a unique path identifier to work concurrently with other processes. The *handle_read_addr_ready* process code works in a similar way to that previously described for the *handle_write_addr_ready* process.

handle_write_data_ready

The *handle_write_data_ready* process handles the WREADY signal for the write data channel. It uses a unique path identifier to work concurrently with other processes.

The *handle_write_data_ready* process code works in a similar way to that previously described for the *handle_write_addr_ready* process.

Chapter 12
Getting Started with Qsys and the BFMs

Note

A license is required to access the Mentor VIP – Intel FPGA Edition Bus Functional Models and Inline Monitor. See "AXI3 BFM Write Data Interleaving" on page 20 for details.

This example shows you how to use the Qsys tool in Quartus Prime software to create a top-level design environment. You will use the *ex1_back_to_back_sv*, a SystemVerilog example from the *$QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi3/qsys-examples* directory in the Intel FPGA Complete Design Suite installation.

You will do the following tasks to set up the design environment:

1. Create a work directory.
2. Copy the example to the work directory.
3. Invoke Qsys from the Quartus Prime software Tools menu.
4. Generate a top-level netlist.
5. Run simulation by referencing the *README* text file and command scripts for your simulation environment.

Setting Up Simulation from a UNIX Platform

The following steps outline how to set up the simulation environment from a UNIX platform.

1. Create a work directory into which you copy the example directory *qsys-examples*, which contains the directory *ex1_back_to_back_sv* from the Installation.

   a. Using the *mkdir* command, create the work directory into which you will copy the *qsys-examples* directory.

   ```
   mkdir axi3-qsys-examples
   ```

   b. Using the *cp* command, copy the *qsys-examples* directory from the *Installation* directory into your work directory.

   ```
   cp -r $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/axi3/\
      qsys-examples/* axi3-qsys-examples/
   ```

2. Using the *cd* command, change the directory path to your local path where the example resides.

```
cd axi3-qsys-examples/ex1_back_to_back_sv
```

3. Open the Qsys tool. Refer to "Running the Qsys Tool" on page 676 for details.

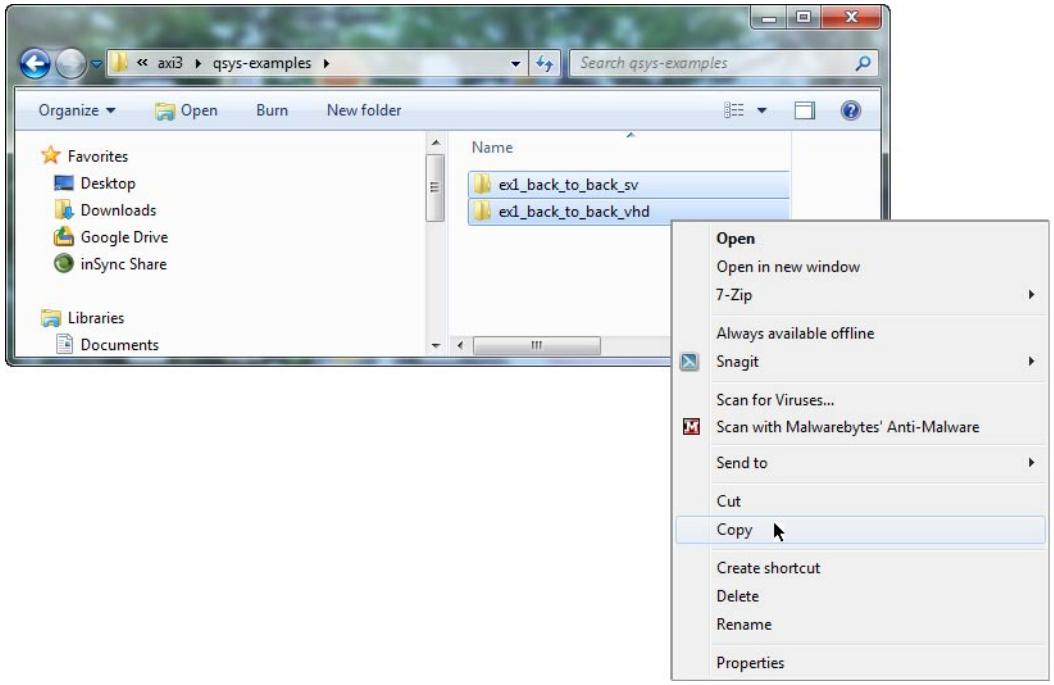
Setting Up Simulation from the Windows GUI

The following steps outline how to set up the simulation environment from a Windows GUI. This example uses the Windows7 platform.

1. Create a work folder into which you copy the contents of the *qsys-examples* folder, which includes the *ex1_back_to_back_sv* folder from the Installation.

   a. Using the GUI, select a location for your work folder, then click the *New folder* option on the window's menu bar to create and name a work folder. For this example, name the work folder *axi3-qsys-examples.* Refer to Figures 12-1 and 12-2 below.

Figure 12-1. Copy the Contents of *qsys-examples* from the Installation Folder



   b. Copy the contents of the *qsys-examples* folder from the *Installation* folder to your work folder.

      Open the *Installation* and work folders. In the *Installation* folder, double-click the *qsys-examples* folder to select and open it. When the folder opens, type `CTRL/A` to

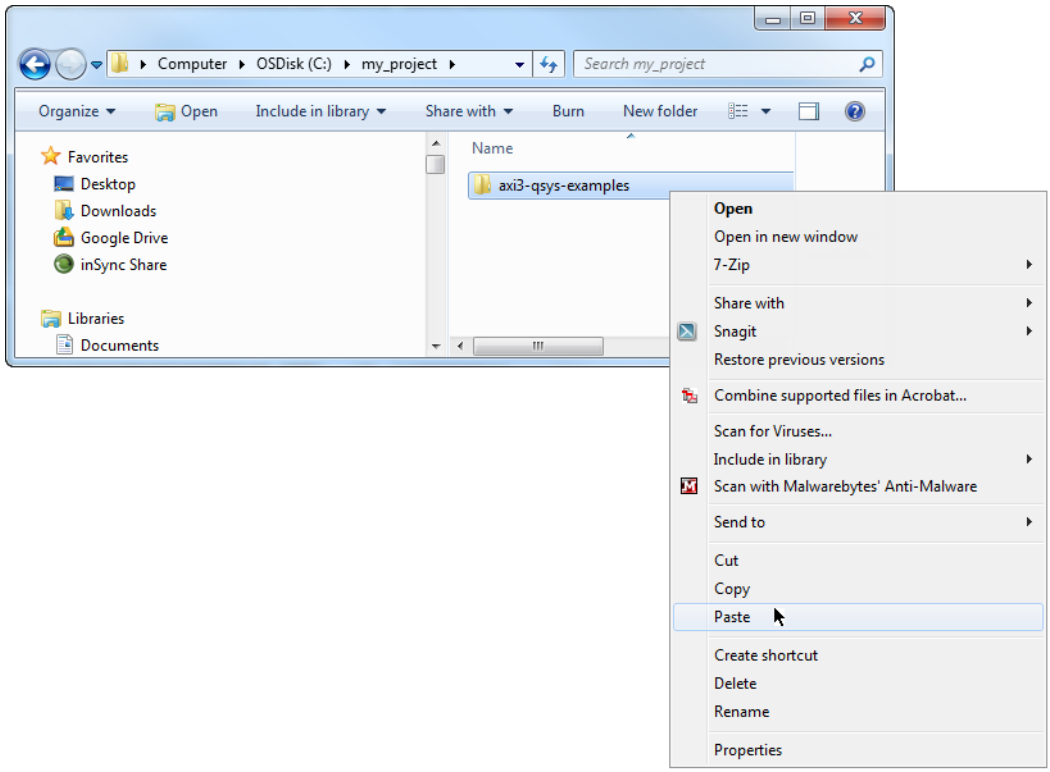
select the contents of the directory, then right-click to display the drop-down menu and select *Copy* from the drop-down menu.

Go to the open work folder. Double-click on the folder.

When the folder opens, right-click inside the work folder and select *Paste* from the drop-down menu to copy the contents of the *qsys-examples* folder to the new *axi3-qsys-examples* work folder.

Paste the *qsys-examples* from the *Installation* folder into the *axi3-qsys-examples* work folder (refer to Figure 12-2).

Figure 12-2. Paste qsys-examples from Installation to Work Folder



Note

Alternatively, open both folders, the *Installation* folder containing the *qsys-examples* folder and the new *axi3-qsys-examples* work folder. Use the Windows *select*, *drag*, and *drop* functions to select the contents of the *qsys-examples* folder in the *Installation* folder, and then drag the contents to and drop it in the new *axi3-qsys-examples* work folder.

2. After creating the new *axi3-qsys-examples* work folder and copying the contents of the *qsys-examples* to it, open the Qsys tool. Refer to “Running the Qsys Tool” on page 676 for details.

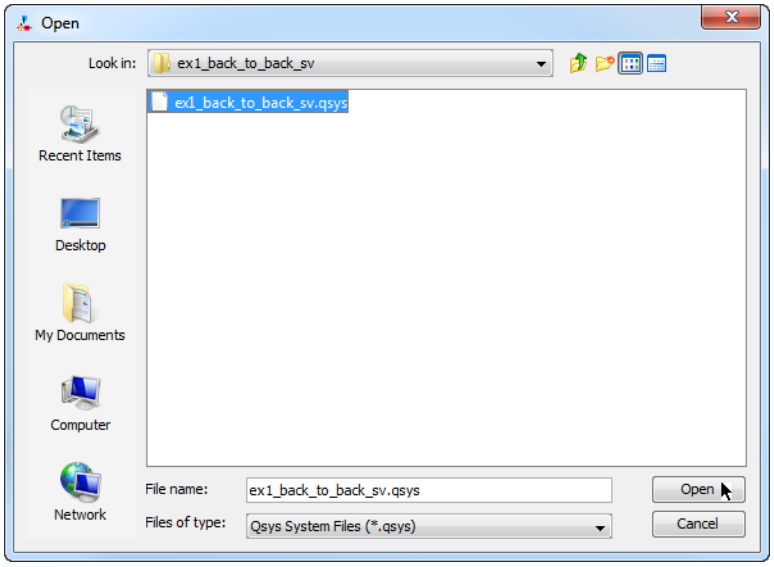
Running the Qsys Tool

1. Open Qsys in the Quartus Prime software menu.
   Start the Quartus Prime software. When the Quartus Prime GUI appears, select *Tools>Qsys*

2. From the Qsys open window, use the *File>Open* command to open and select the file *ex1_back_to_back_sv.qsys.* This Qsys file is in the directory *axi3-qsys-examples/ex1_back_to_back_sv* (refer to Figure 12-3).

   Select and open the *ex1_back_to_back_sv.qsys* example.

Figure 12-3. Open the *ex1_back_to_back_sv.qsys* Example

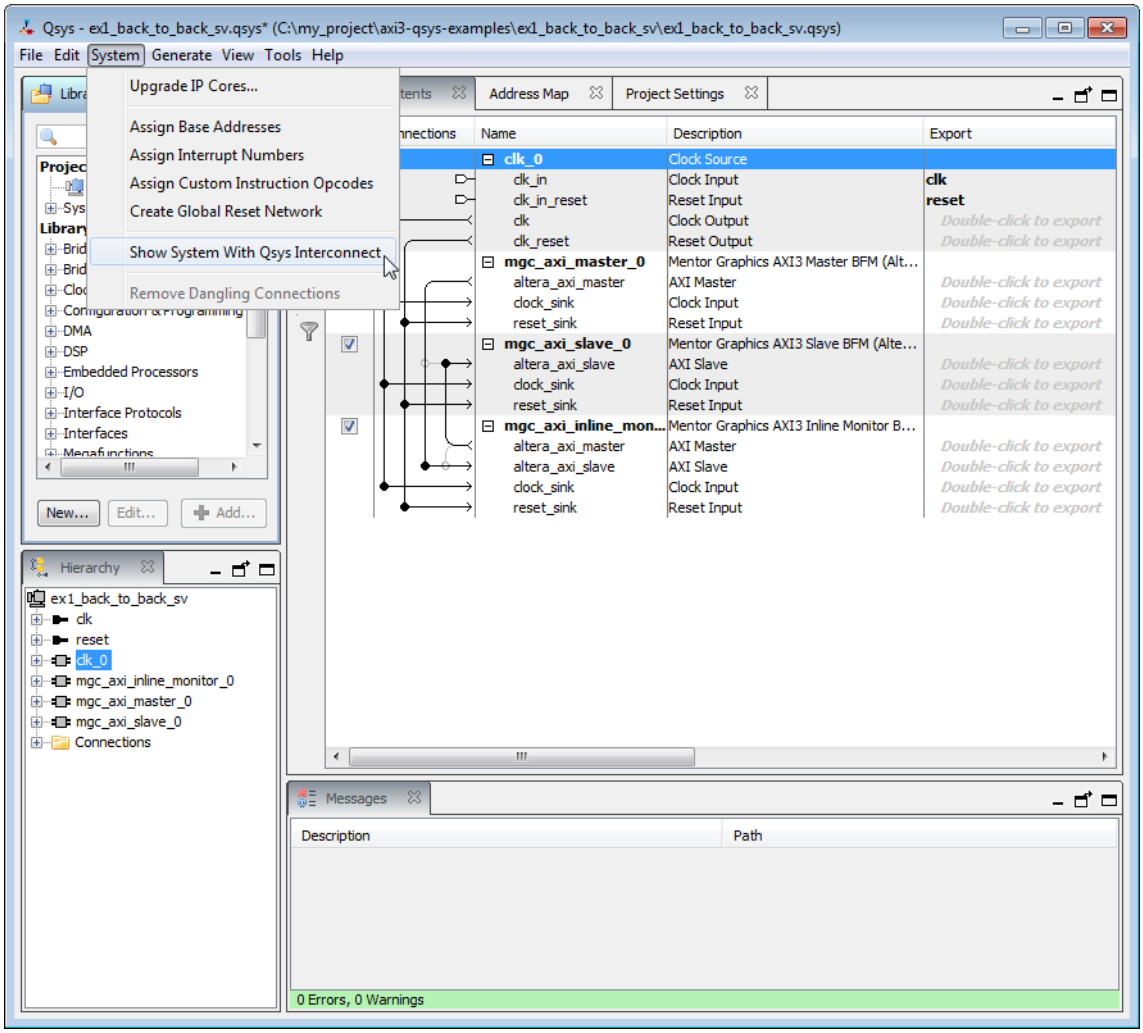


Note

If you open the Qsys tool in a subsequent session, a Qsys dialog asks you if you want to open this file.

3. Qsys displays the connectivity of the selected example as shown in Figure 12-4.

Figure 12-4. Show System With Qsys Interconnect



Note

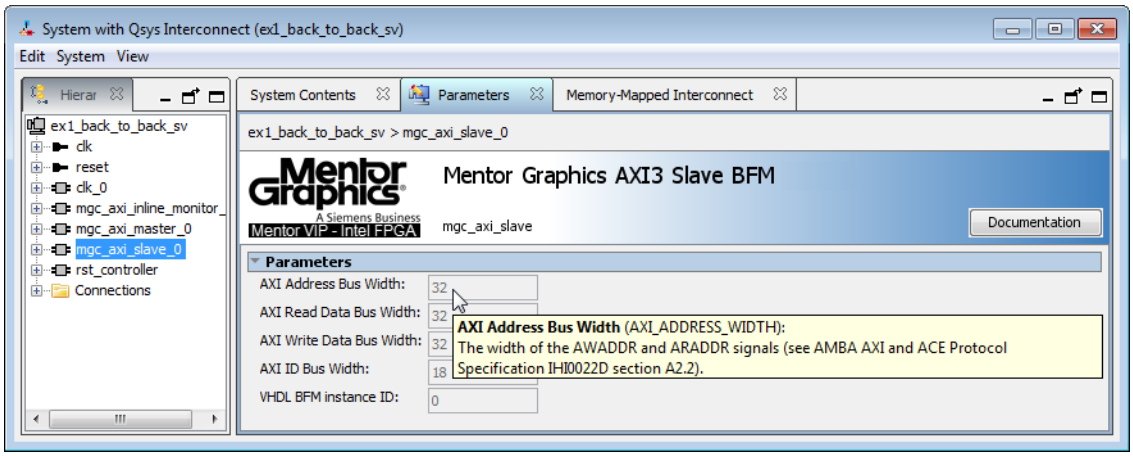
If you are using VHDL, you must select each BFM and verify that the index number specified for the BFM is correct. An information dialog displays the properties of the BFM when you select it. Ensure that the specified BFM *index* is correct in this dialog. If you do not know the correct index number, check the VHDL code for the BFM.

4. Click the *System* drop-down menu on the main Qsys toolbar, and select *Show System With Qsys Interconnect* to open the System With Qsys Interconnect options window, as shown in Figure 12-5.
5. Select a BFM within the Hierarchy pane of the System With Qsys Interconnect window, in this case the *mgc_axi3_slave_0*. Click the Parameters tab to reveal the parameter editor to review and change the selected BFM parameters.

Note

Placing the mouse pointer over a parameter name, or its value, opens a documentation popup for the parameter. Parameter documentation is also available by clicking the Documentation button.

Figure 12-5. System With Qsys Interconnect Parameters Tab



Note

You can configure the *USE_** parameters, which enable and disable optional AXI4 BFM signals, using the Parameter Editor.

6. Close the System With Qsys Interconnect window after your parameter edits are complete.
7. Click the *Generate* drop-down menu on the main Qsys toolbar, and select *Generate HDL* to open the Generation options window.
8. Specify the Generation window options, as shown in Figure 12-6.
   a. Synthesis section
      i. Set the *Create HDL design files for synthesis* to *None* to inhibit the generation of synthesis files.
      ii. Uncheck the *Create block symbol file (.bsf)* check box.

b. Simulation section

   i. Set the *Create simulation model* to *Verilog.*

c. Change the path of the example. In the *Path* field of the Output Directory section, ensure the path correctly specifies the subdirectory *ex1_back_to_back_sv*, which is the subdirectory containing the example that you just copied into a temporary directory.

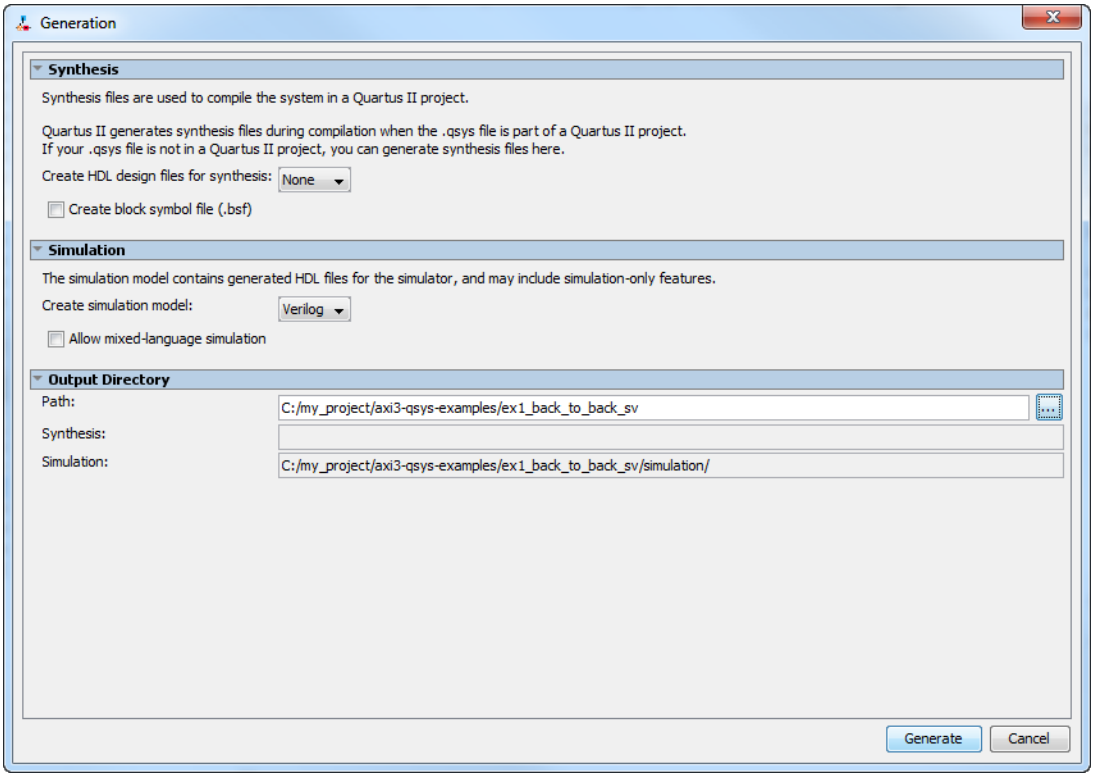
Note

If the subdirectory name of the example is duplicated in the *Path* field, you must remove one of the duplicated subdirectory names. To reset the path, double-click the square browse button to the right of the *Path* field and locate the correct path of the example.

The path name of the example specified in the *Path* field of the Output Directory section **must be correct before** generating the HDL for the example.

9. Click the *Generate* button on the bottom right side of the window, as shown in Figure 12-6.

Figure 12-6. Qsys Generation Window Options



10. Refer to "Running a Simulation" on page 680 to start simulation.

Running a Simulation

Note

Throughout this section, substitute *axi3-qsys-examples* with *axi4-qsys-examples*, and *axi* or *axi3* with *axi4* for the AXI4 protocol.

The choice of simulator determines the process that you follow to run a simulation. The process for each simulator is detailed in the following sections:

- "ModelSim Simulation" on page 680
- "Questa Simulation" on page 684
- "Cadence Xcelium Simulation" on page 684
- "Synopsys VCS Simulation" on page 686

For each simulator, a *README* text file and a command script file is provided in the installed Mentor VIP – Intel FPGA Edition directory location *axi3/qsys-examples/ex1_back_to_back_sv*.

Table 12-1 details the *README* text file instructions to load a model into the simulator, and the script command file to start the simulation.

Table 12-1. SystemVerilog README Files and Script Names for all Simulators

	Questa Simulation	ModelSim Simulation	Xcelium Simulation	VCS Simulation
README	README-Questa.txt	README-ModelSim.txt	README-IUS.txt	README-VCS.txt
Script File	example.do	example.do	example-ius.sh	example-vcs.sh

Note

The VHDL example *axi3/qsys-examples/ex1_back_to_back_vhd* has equivalent *README* text files and command script files. The process to follow for VHDL simulation is similar to that for SystemVerilog simulation.

Note

The Mentor VIP - Intel FPGA Edition 2019.1 release and above support the use of multiple components from different library releases in the same test bench. The simulator can use the relevant *.so* files for this support.

ModelSim Simulation

You can run a ModelSim simulation from a GUI interface or a command line. Before starting a simulation, you must do the following:

- Check that the *$QUARTUS_ROOTDIR* environment variable points to the Quartus Prime software directory in the Quartus Prime software installation. The example command script *example.do* requires this variable to locate the installed Mentor VIP – Intel FPGA Edition BFMs during simulation.
- Ensure that the environment variable *MvcHome* points to the location of the installed BFM. You can set the location of *MvcHome* using one of the following options:
  - To set the *MvcHome* variable in the *modelsim.ini* file, refer to "Editing the modelsim.ini File" on page 683."
  - To specify the *-mvchome* option on the command line, refer to "Starting a Simulation from a UNIX Command Line" on page 682."

The following sections outline how to run a ModelSim simulation from either a GUI or a command line.

Starting a Simulation from the ModelSim GUI

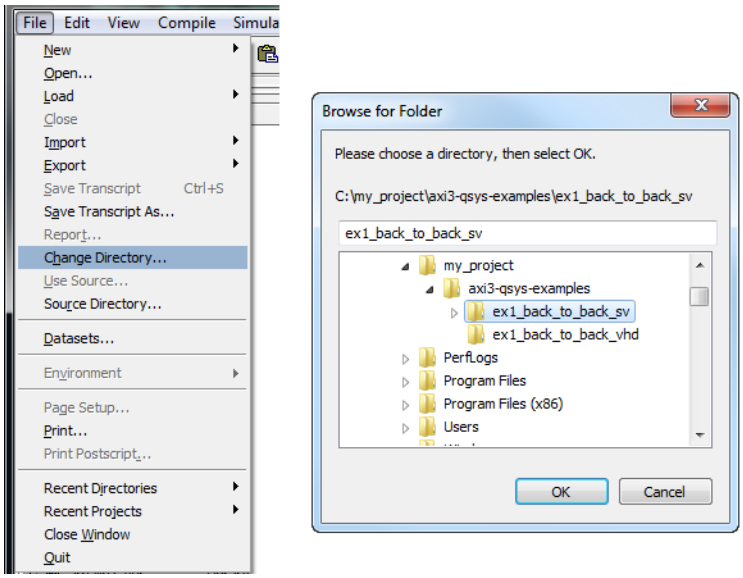
To start a simulation with the ModelSim simulator GUI:

1. Start the ModelSim GUI.

```
vsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
```

2. Change the directory to the work directory that contains the example to be simulated with method (a) or (b) below.
   a. From the *File* menu, click the *Change Directory* option. When the *Browse for Folder* dialog appears, select the work directory that contains the example.

Figure 12-7. Select the Work Directory



b. In the ModelSim Transcript window, change to the work directory containing the example to simulate:

```
vsim> cd axi3-qsys-examples/ex1_back_to_back_sv
```

3. Run the *example.do* script within the Transcript window by typing the following command to compile and elaborate the test programs:

```
vsim> do example.do
```

Note

For details about the processing performed by the *example.do* script, refer to "ModelSim Example Script Processing" on page 682.

4. In the Transcript window, start the simulation and run to completion:

```
vsim> run -all
```

Starting a Simulation from a UNIX Command Line

To start a simulation with the ModelSim simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated.

```
cd axi3-qsys-examples/ex1_back_to_back_sv
```

2. In a shell, start the Modelsim simulator with the *example.do* script.

```
vsim -mvchome $QUARTUS_ROOTDIR/../ip/altera/\
      mentor_vip_ae/common -gui -do example.do
```

Note

For details about the processing performed by the *example.do* script, refer to "ModelSim Example Script Processing" on page 682.

3. In the Transcript window, start the simulation and run to completion.

```
vsim> run -all
```

ModelSim Example Script Processing

The *example.do* script described below is contained in the installed Mentor VIP – Intel FPGA Edition directory location *axi3/qsys-examples/ex1_back_to_back_sv*.

The Mentor VIP – Intel FPGA Edition BFMs for AXI3 are compiled.

```
set TOP_LEVEL_NAME top
set QSYS_SIMDIR    simulation

source $QSYS_SIMDIR/mentor/msim_setup.tcl
if {![info exists env(MENTOR_VIP_AE)]}
{
  set env(MENTOR_VIP_AE) $env(QUARTUS_ROOTDIR)/../ip/altera/mentor_vip_ae
}

   ensure_lib libraries
   ensure_lib libraries/work
   vmap work  libraries/work

   vlog -work work -sv \
      $env(MENTOR_VIP_AE)/common/questa_mvc_svapi.svh \
      $env(MENTOR_VIP_AE)/axi3/bfm/mgc_common_axi.sv \
      $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_monitor.sv \
      $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_inline_monitor.sv \
      $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_master.sv \
      $env(MENTOR_VIP_AE)/axi3/bfm/mgc_axi_slave.sv
```

The two *tcl* alias commands *dev_com* and *com* compile the required design files. These alias commands are defined in the *msim_setup.tcl* simulation script generated by Qsys, along with the simulation model files:

```
# Compile device library files
dev_com

# Compile Qsys-generated design files
com
```

The three example test programs are compiled:

```
# Compile example test program files
vlog  master_test_program.sv
vlog   slave_test_program.sv
vlog monitor_test_program.sv
```

The example top-level file is compiled:

```
# Compile top-level design file
vlog top.sv
```

Simulation starts with the *elab* alias defined in the *msim_setup.tcl* simulation script generated by Qsys:

```
# Simulate
elab
```

Editing the modelsim.ini File

The ModelSim simulator does not have a default installation directory path defined for the environment variable *MvcHome*; therefore, you must define a path for this variable.

Note

Setting *MvcHome* within the *modelsim.ini* file eliminates the need to specify the *-mvchome* option on the *vsim* command line.

To provide the installation directory path of the Mentor VIP – Intel FPGA Edition for running a ModelSim simulation:

1. Edit the *modelsim.ini* file and find the section that starts with *[vsim]*.
2. Search for *MvcHome*. If it is not already defined in the *modelsim.ini* file, you must add it. You can add this variable at any location in the *[vsim]* section.

   If the *modelsim.ini* file is read-only, you must modify the permissions of the file to allow write access.
3. Add or change the *MvcHome* path to point to the location where the Mentor VIP – Intel FPGA Edition is installed. Do not forget the *common* subdirectory.

```
MvcHome = $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common
```

Note

Do not use the ModelSim *vmap* command to specify the installed location of the Mentor VIP – Intel FPGA Edition because this places the definition of the environment variable *MvcHome* in the *[library]* section of *modelsim.ini*. For example, do not use the command `vmap MvcHome $QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae/common`.

Questa Simulation

To run a Questa simulation, follow the process detailed in "ModelSim Simulation" on page 680.

Cadence Xcelium Simulation

Before starting a Cadence Xcelium simulation, you must do the following:

- Check that the *$QUARTUS_ROOTDIR* environment variable points to the Quartus Prime software directory in the Quartus Prime software installation. The example script *example-ius.sh* requires this variable to locate the Mentor VIP – Intel FPGA Edition BFMs during simulation.
- Set the environment variable *CDS_ROOT* to the installation directory of the Xcelium Verilog compiler *ncvlog*. The *cds_root* command returns the installation directory of the specified tool *ncvlog*.

```
setenv CDS_ROOT        `cds_root ncvlog`
```

Starting a Simulation from a UNIX Command Line

To start a simulation with the Cadence Xcelium simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated:

   ```
   cd axi3-qsys-examples/ex1_back_to_back_sv
   ```

2. Start the Cadence Xcelium simulator with the *example-ius.sh* script.
   - For a 32-bit simulation, execute this command:

     ```
     sh example-ius.sh 32
     ```

   - For a 64-bit simulation, execute this command:

     ```
     sh example-ius.sh 64
     ```

Note

For details about the process steps performed by the *example-ius.sh* script, refer to "Cadence Xcelium Example Script Processing" on page 685.

Cadence Xcelium Example Script Processing

The *example-ius.sh* script described below is contained in the installed Mentor VIP – Intel FPGA Edition directory location *axi3/qsys-examples/ex1_back_to_back_sv*.

The Mentor VIP – Intel FPGA Edition BFMs for AXI3 are compiled. The *ncsim_setup.sh* simulation script is generated by Qsys, along with the simulation model files.

```
#!/bin/sh

# Usage: <command> [32|64]
# 32 bit mode is run unless 64 is passed in as the first argument.

MENTOR_VIP_AE=${MENTOR_VIP_AE:-$QUARTUS_ROOTDIR/../ip/ \
                              altera/mentor_vip_ae}

if [ "$1" == "64" ]
then
   export QUESTA_MVC_GCC_LIB=$MENTOR_VIP_AE/common/ \
         questa_mvc_core/linux_x86_64_gcc-4.4_ius
   export INCA_64BIT=1
else
   export QUESTA_MVC_GCC_LIB=$MENTOR_VIP_AE/common/ \
         questa_mvc_core/linux_gcc-4.4_ius
fi
export LD_LIBRARY_PATH=$QUESTA_MVC_GCC_LIB:$LD_LIBRARY_PATH

cd simulation/cadence
# Run once, just to execute the 'mkdir' for the libraries.
source ncsim_setup.sh SKIP_DEV_COM=1 SKIP_COM=1 SKIP_ELAB=1 SKIP_SIM=1
```

```
# Compile VIP
   ncvlog -sv \
      "$MENTOR_VIP_AE/common/questa_mvc_svapi.svh" \
      "$MENTOR_VIP_AE/axi3/bfm/mgc_common_axi.sv" \
      "$MENTOR_VIP_AE/axi3/bfm/mgc_axi_monitor.sv" \
      "$MENTOR_VIP_AE/axi3/bfm/mgc_axi_inline_monitor.sv" \
      "$MENTOR_VIP_AE/axi3/bfm/mgc_axi_master.sv" \
      "$MENTOR_VIP_AE/axi3/bfm/mgc_axi_slave.sv"
```

The three example test programs are compiled:

```
# Compile the test program
ncvlog -sv ../../master_test_program.sv
ncvlog -sv ../../monitor_test_program.sv
ncvlog -sv ../../slave_test_program.sv
```

The example top-level file is compiled:

```
# Compile the top
ncvlog -sv ../../top.sv
```

Elaboration and simulation starts with the *ncsim_setup.sh* command. The Cadence Xcelium simulator requires the SystemVerilog library path *-sv_lib* to be passed to the simulator.

```
# Elaborate and simulate
source ncsim_setup.sh \
   USER_DEFINED_ELAB_OPTIONS="\"-timescale 1ns/1ns\"" \
   USER_DEFINED_SIM_OPTIONS="\"-MESSAGES \
      -sv_lib $QUESTA_MVC_GCC_LIB/libaxi_IN_SystemVerilog_IUS_full_DVC\""
\
   TOP_LEVEL_NAME=top
```

Synopsys VCS Simulation

Before starting a Synopsys VCS simulation, you must do the following:

- Ensure that the *$QUARTUS_ROOTDIR* environment variable points to the Quartus Prime software directory in the Quartus Prime software installation. The example script *example-vcs.sh* requires this variable to locate the Mentor VIP – Intel FPGA Edition BFMs during simulation.
- Set the environment variable *VCS_HOME* to the installation directory of the VCS Verilog compiler.

  ```
  setenv VCS_HOME <Installation-of-VCS>
  ```

Starting a Simulation from a UNIX Command Line

To start a simulation with the Synopsys VCS simulator from a UNIX command line:

1. Change the directory to the work directory containing the example to be simulated:

```
cd axi3-qsys-examples/ex1_back_to_back_sv
```

2. Start the Synopsys VCS simulator with the *example-vcs.sh* script. F
   - For a 32-bit simulation, execute this command:

     ```
     sh example-vcs.sh 32
     ```

   - For a 64-bit simulation, execute this command:

     ```
     sh example-vcs.sh 64
     ```

Note

For details about the process steps performed by the *example-vcs.sh* script, refer to "Synopsys VCS Example Script Processing" on page 687.

Synopsys VCS Example Script Processing

The *example-vcs.sh* script described below is contained in the installed Mentor VIP – Intel FPGA Edition directory location *axi3/qsys-examples/ex1_back_to_back_sv*.

The Mentor VIP – Intel FPGA Edition BFMs for AXI3 are compiled. The *vcs_setup.sh* simulation script is generated by Qsys, along with the simulation model files.

```
#!/bin/sh

# Usage: <command> [32|64]
# 32 bit mode is run unless 64 is passed in as the first argument.

MENTOR_VIP_AE=${MENTOR_VIP_AE:-
$QUARTUS_ROOTDIR/../ip/altera/mentor_vip_ae}

if [ "$1" == "64" ]
then
   export RUN_64bit=-full64
   export VCS_TARGET_ARCH=`getsimarch 64`
   export LD_LIBRARY_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_64-shared/lib64
   export QUESTA_MVC_GCC_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_64-shared
   export QUESTA_MVC_GCC_LIB=${MENTOR_VIP_AE}/common/ \
                          questa_mvc_core/linux_x86_64_gcc-4.7.2_vcs
else
   export RUN_64bit=
   export LD_LIBRARY_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_32-shared/lib
   export QUESTA_MVC_GCC_PATH=${VCS_HOME}/gnu/linux/gcc-4.7.2_32-shared
   export QUESTA_MVC_GCC_LIB=${MENTOR_VIP_AE}/common/ \
                          questa_mvc_core/linux_gcc-4.7.2_vcs
fi

cd simulation/synopsys/vcs
rm -rf csrc simv simv.daidir transcript ucli.key vc_hdrs.h

# VCS accepts the -LDFLAGS flag on the command line, but the shell quoting
# is too difficult. Just set the LDFLAGS ENV variable for the compiler to
# pick up. Alternatively, use the VCS command line option '-file' with the
```

```
# LDFLAGS set (this avoids shell quoting issues).
# vcs-switches.f:
# -LDFLAGS "-L ${QUESTA_MVC_GCC_LIB} -Wl,-rpath ${QUESTA_MVC_GCC_LIB}
# -laxi_IN_SystemVerilog_VCS_full_DVC"
export LDFLAGS="-L ${QUESTA_MVC_GCC_LIB} -Wl, \
-rpath ${QUESTA_MVC_GCC_LIB} -laxi_IN_SystemVerilog_VCS_full_DVC"

USER_DEFINED_ELAB_OPTIONS="\"\
   $RUN_64bit \
   +systemverilogext+.sv +vpi +acc +vcs+lic+wait \
   -cpp ${QUESTA_MVC_GCC_PATH}/xbin/g++ \
   \
   $MENTOR_VIP_AE/common/questa_mvc_svapi.svh \
   $MENTOR_VIP_AE/axi3/bfm/mgc_common_axi.sv \
   $MENTOR_VIP_AE/axi3/bfm/mgc_axi_monitor.sv \
   $MENTOR_VIP_AE/axi3/bfm/mgc_axi_inline_monitor.sv \
   $MENTOR_VIP_AE/axi3/bfm/mgc_axi_slave.sv \
   $MENTOR_VIP_AE/axi3/bfm/mgc_axi_master.sv \
   \
```

The three example test programs and top-level file are compiled:

```
   ../../../master_test_program.sv \
   ../../../monitor_test_program.sv  \
   ../../../slave_test_program.sv \
   ../../../top.sv  \""
```

Elaboration and simulation starts with the *vcs_setup.sh* command:

```
source vcs_setup.sh \
   USER_DEFINED_ELAB_OPTIONS="$USER_DEFINED_ELAB_OPTIONS" \
   USER_DEFINED_SIM_OPTIONS="'-l transcript'" \
   TOP_LEVEL_NAME=top
```

Appendix A

AXI3 Assertions

The AXI3 master, slave, and monitor BFMs all support error checking with the firing of one or more assertions when a property defined in the AMBA AXI Protocol Specification has been violated. Each assertion can be individually enabled/disabled using the *set_config()* function for a particular BFM. The property covered for each assertion is noted in Table A-1 under the Property Reference column. The reference number refers to the section number in the AMBA AXI Protocol Specification.

Table A-1. AXI3 Assertions

Error Code	Error Name	Description	Property Ref
AXI3-60000	AXI_ARESETn_SIGNAL_Z	ARESETn has a Z value.	
AXI3-60001	AXI_ARESETn_SIGNAL_X	ARESETn has an X value.	
AXI3-60002	AXI_ACLK_SIGNAL_Z	ACLK has a Z value.	
AXI3-60003	AXI_ACLK_SIGNAL_X	ACLK has an X value.	
AXI3-60004	AXI_ADDR_FOR_READ_BURST_ACROSS_4K_BOUNDARY	This read transaction has crossed a 4KB boundary.	A3.4.1
AXI3-60005	AXI_ADDR_FOR_WRITE_BURST_ACROSS_4K_BOUNDARY	This write transaction has crossed a 4KB boundary.	A3.4.1
AXI3-60006	AXI_ARADDR_CHANGED_BEFORE_ARREADY	The value of ARADDR has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60007	AXI_ARADDR_UNKN	ARADDR has an X or Z value.	A2.5
AXI3-60008	AXI_ARBURST_CHANGED_BEFORE_ARREADY	The value of ARBURST has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60009	AXI_ARBURST_UNKN	ARBURST has an X or Z value.	A2.5

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60010	AXI_ARCACHE_CHANGED_BEFORE_ARREADY	The value of ARCACHE has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60011	AXI_ARCACHE_UNKN	ARCACHE has an X or Z value.	A2.5
AXI3-60012	AXI_ARID_CHANGED_BEFORE_ARREADY	The value of ARID has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60013	AXI_ARID_UNKN	ARID has an X or Z value.	A2.5
AXI3-60014	AXI_ARLEN_CHANGED_BEFORE_ARREADY	The value of ARLEN has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60015	AXI_ARLEN_UNKN	ARLEN has an X or Z value.	A2.5
AXI3-60016	AXI_ARLOCK_CHANGED_BEFORE_ARREADY	The value of ARLOCK has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60017	AXI_ARLOCK_UNKN	ARLOCK has an X or Z value.	A2.5
AXI3-60018	AXI_ARPROT_CHANGED_BEFORE_ARREADY	The value of ARPROT has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60019	AXI_ARPROT_UNKN	ARPROT has an X or Z value.	A2.5
AXI3-60020	AXI_ARREADY_UNKN	ARREADY has an X or Z value.	A2.5
AXI3-60021	AXI_ARSIZE_CHANGED_BEFORE_ARREADY	The value of ARSIZE has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1
AXI3-60022	AXI_ARSIZE_UNKN	ARSIZE has an X or Z value.	A2.5
AXI3-60023	AXI_ARUSER_CHANGED_BEFORE_ARREADY	The value of ARUSER has changed from its initial value between the time ARVALID was asserted, and before ARREADY was asserted.	A3.2.1

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60024	AXI_ARUSER_UNKN	ARUSER has an X or Z value.	A2.5
AXI3-60025	AXI_ARVALID_DEASSERTED_BEFORE_ARREADY	ARVALID has been de-asserted before ARREADY was asserted.	A3.2.1
AXI3-60026	AXI_ARVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A master interface must begin driving ARVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI3-60027	AXI_ARVALID_UNKN	ARVALID has an X or Z value.	A2.5
AXI3-60028	AXI_AWADDR_CHANGED_BEFORE_AWREADY	The value of AWADDR has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1
AXI3-60029	AXI_AWADDR_UNKN	AWADDR has an X or Z value.	A2.2
AXI3-60030	AXI_AWBURST_CHANGED_BEFORE_AWREADY	The value of AWBURST has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1
AXI3-60031	AXI_AWBURST_UNKN	AWBURST has an X or Z value.	A2.2
AXI3-60032	AXI_AWCACHE_CHANGED_BEFORE_AWREADY	The value of AWCACHE has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1
AXI3-60033	AXI_AWCACHE_UNKN	AWCACHE has an X or Z value.	A2.2
AXI3-60034	AXI_AWID_CHANGED_BEFORE_AWREADY	The value of AWID has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted (SPEC3(3.1))	A3.2.1
AXI3-60035	AXI_AWID_UNKN	AWID has an X or Z value.	A2.2
AXI3-60036	AXI_AWLEN_CHANGED_BEFORE_AWREADY	The value of AWLEN has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1
AXI3-60037	AXI_AWLEN_UNKN	AWLEN has an X or Z value.	A2.2
AXI3-60038	AXI_AWLOCK_CHANGED_BEFORE_AWREADY	The value of AWLOCK has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60039	AXI_AWLOCK_UNKN	AWLOCK has an X or Z value.	A2.2
AXI3-60040	AXI_AWPROT_CHANGED_BEFORE_AWREADY	The value of AWPROT has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1
AXI3-60041	AXI_AWPROT_UNKN	AWPROT has an X or Z value.	A2.2
AXI3-60042	AXI_AWREADY_UNKN	AWREADY has an X or Z value.	A2.2
AXI3-60043	AXI_AWSIZE_CHANGED_BEFORE_AWREADY	The value of AWSIZE has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1
AXI3-60044	AXI_AWSIZE_UNKN	AWSIZE has an X or Z value.	A2.2
AXI3-60045	AXI_AWUSER_CHANGED_BEFORE_AWREADY	The value of AWUSER has changed from its initial value between the time AWVALID was asserted, and before AWREADY was asserted.	A3.2.1
AXI3-60046	AXI_AWUSER_UNKN	AWUSER has an X or Z value.	A2.2
AXI3-60047	AXI_AWVALID_DEASSERTED_BEFORE_AWREADY	AWVALID has been de-asserted before AWREADY was asserted.	A3.2.1
AXI3-60048	AXI_AWVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A master interface must begin driving AWVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI3-60049	AXI_AWVALID_UNKN	AWVALID has an X or Z value.	A2.2
AXI3-60050	AXI_BID_CHANGED_BEFORE_BREADY	The value of BID has changed from its initial value between the time BVALID was asserted, and before BREADY was asserted.	A3.2.1
AXI3-60051	AXI_BID_UNKN	BID has an X or Z value.	A2.4
AXI3-60052	AXI_BREADY_UNKN	BREADY has an X or Z value.	A2.4
AXI3-60053	AXI_BRESP_CHANGED_BEFORE_BREADY	The value of BRESP has changed from its initial value between the time BVALID was asserted, and before BREADY was asserted.	A3.2.1
AXI3-60054	AXI_BRESP_UNKN	BRESP has an X or Z value.	A2.4

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60055	AXI_BUSER_CHANGED_BEFORE_BREADY	The value of BUSER has changed from its initial value between the time BVALID was asserted, and before BREADY was asserted.	A3.2.1
AXI3-60056	AXI_BUSER_UNKN	BUSER has an X or Z value.	A2.4
AXI3-60057	AXI_BVALID_DEASSERTED_BEFORE_BREADY	BVALID has been de-asserted before BREADY was asserted.	A3.2.1
AXI3-60058	AXI_BVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A slave interface must begin driving BVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI3-60059	AXI_BVALID_UNKN	BVALID has an X or Z value.	A2.4
AXI3-60060	AXI_EXCLUSIVE_READ_ACCESS_MODIFIABLE	The modifiable bit (bit 1 of the cache transaction field) should not be set for an exclusive read access.	A7.2.4
AXI3-60061	AXI_EXCLUSIVE_READ_BYTES_TRANSFER_EXCEEDS_128	Number of bytes in an exclusive read transaction must be less than or equal to 128.	A7.2.4
AXI3-60062	AXI_EXCLUSIVE_WRITE_BYTES_TRANSFER_EXCEEDS_128	Number of bytes in an exclusive write transaction must be less than or equal to 128.	A7.2.4
AXI3-60063	AXI_EXCLUSIVE_READ_BYTES_TRANSFER_NOT_POWER_OF_2	Number of bytes of an exclusive read transaction is not a power of 2.	A7.2.4
AXI3-60064	AXI_EXCLUSIVE_WRITE_BYTES_TRANSFER_NOT_POWER_OF_2	Number of bytes of an exclusive write transaction is not a power of 2.	A7.2.4
AXI3-60065	AXI_EXCLUSIVE_WR_ADDRESS_NOT_SAME_AS_RD	Exclusive write does not match the address of the previous exclusive read to this id.	A7.2.4
AXI3-60066	AXI_EXCLUSIVE_WR_BURST_NOT_SAME_AS_RD	Exclusive write does not match the burst setting of the previous exclusive read to this id.	A7.2.4
AXI3-60067	AXI_EXCLUSIVE_WR_CACHE_NOT_SAME_AS_RD	Exclusive write does not match the cache setting of the previous exclusive read to this id.	A7.2.4
AXI3-60068	AXI_EXCLUSIVE_WRITE_ACCESS_MODIFIABLE	The modifiable bit (bit 1 of the cache transaction field) should not be set for an exclusive write access.	A7.2.4

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60069	AXI_EXCLUSIVE_WR_LENGTH_NOT_SAME_AS_RD	Exclusive write does not match the length of the previous exclusive read to this id.	A7.2.4
AXI3-60070	AXI_EXCLUSIVE_WR_PROT_NOT_SAME_AS_RD	Exclusive write does not match the prot setting of the previous exclusive read to this id.	A7.2.4
AXI3-60071	AXI_EXCLUSIVE_WR_SIZE_NOT_SAME_AS_RD	Exclusive write does not match the size of the previous exclusive read to this id.	A7.2.4
AXI3-60072	AXI_EXOKAY_RESPONSE_NORMAL_READ	Slave has responded AXI_EXOKAY to a nonexclusive read transfer.	-
AXI3-60073	AXI_EXOKAY_RESPONSE_NORMAL_WRITE	Slave has responded AXI_EXOKAY to a nonexclusive write transfer.	-
AXI3-60074	AXI_EX_RD_RESP_MISMATCHED_WITH_EXPECTED_RESP	Expected response to this exclusive read did not matched with the actual response.	A7.2.3
AXI3-60075	AXI_EX_WR_RESP_MISMATCHED_WITH_EXPECTED_RESP	Expected response to this exclusive write did not matched with the actual response.	A7.2.3
AXI3-60076	AXI_EX_RD_EXOKAY_RESP_SLAVE_WITHOUT_EXCLUSIVE_ACCESS	Response for an exclusive read to a slave which does not support exclusive access should be AXI_OKAY, but it returned AXI_EXOKAY.	A7.2.3
AXI3-60077	AXI_EX_WRITE_BEFORE_EX_READ_RESPONSE	Exclusive write has occurred, with no previous exclusive read.	A7.2.2
AXI3-60078	AXI_EX_WRITE_EXOKAY_RESP_SLAVE_WITHOUT_EXCLUSIVE_ACCESS	Response for an exclusive write to a slave which does not support exclusive access should be AXI_OKAY, but it returned AXI_EXOKAY.	A7.2.3
AXI3-60079	AXI_ILLEGAL_LENGTH_WRAPPING_READ_BURST	In the last read address phase burst_length has an illegal value for a burst of type AXI_WRAP.	A3.4.1
AXI3-60080	AXI_ILLEGAL_LENGTH_WRAPPING_WRITE_BURST	In the last write address phase burst_length has an illegal value for a burst of type AXI_WRAP.	A3.4.1
AXI3-60081	AXI_ILLEGAL_RESPONSE_EXCLUSIVE_READ	Response for an exclusive read should be either AXI_OKAY or AXI_EXOKAY.	A7.2.3
AXI3-60082	AXI_ILLEGAL_RESPONSE_EXCLUSIVE_WRITE	Response for an exclusive write should be either AXI_OKAY or AXI_EXOKAY.	A7.2.3
AXI3-60083	AXI_PARAM_READ_DATA_BUS_WIDTH	The value of AXI_RDATA_WIDTH must be one of 8,16,32,64,128,256,512,1024.	A1.3.1
AXI3-60084	AXI_PARAM_WRITE_DATA_BUS_WIDTH	The value of AXI_WDATA_WIDTH must be one of 8,16,32,64,128,256,512,1024.	A1.3.1

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60085	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_12	The RA bit of the cache transaction field should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60086	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_13	The RA bit of the cache transaction field should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60087	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_4	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60088	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_5	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60089	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_8	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60090	AXI_READ_ALLOCATE_WHEN_NON_MODIFIABLE_9	The RA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60091	AXI_READ_BURST_LENGTH_VIOLATION	The number of beats actually read does not match the burst length defined by the ARLEN.	A3.4.1
AXI3-60092	AXI_READ_BURST_SIZE_VIOLATION	In this read transaction, size has been set greater than the defined data bus.	A3.4.1
AXI3-60093	AXI_READ_DATA_BEFORE_ADDRESS	An unexpected read response has occurred (there are no outstanding read transactions with this id).	A3.3.1
AXI3-60094	AXI_READ_DATA_CHANGED_BEFORE_RREADY	The value of RDATA has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	A3.2.1
AXI3-60095	AXI_READ_DATA_UNKN	RDATA has an X or Z value.	A2.6
AXI3-60096	AXI_RESERVED_ARLOCK_ENCODING	The reserved encoding of 2'b11 should not be used for ARLOCK.	A7.4
AXI3-60097	AXI_READ_RESP_CHANGED_BEFORE_RREADY	The value of RRESP has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	A3.2.1
AXI3-60098	AXI_RESERVED_ARBURST_ENCODING	The reserved encoding of 2'b11 should not be used for ARBURST.	A3.4.1
AXI3-60099	AXI_RESERVED_AWBURST_ENCODING	The reserved encoding of 2'b11 should not be used for AWBURST.	A3.4.1

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60100	AXI_RID_CHANGED_BEFORE_RREADY	The value of RID has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	A3.2.1
AXI3-60101	AXI_RID_UNKN	RID has an X or Z value.	A2.6
AXI3-60102	AXI_RLAST_CHANGED_BEFORE_RREADY	The value of RLAST has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	A3.2.1
AXI3-60103	AXI_RLAST_UNKN	RLAST has an X or Z value.	A2.6
AXI3-60104	AXI_RREADY_UNKN	RREADY has an X or Z value.	A2.6
AXI3-60105	AXI_RRESP_UNKN	RRESP has an X or Z value.	A2.6
AXI3-60106	AXI_RUSER_CHANGED_BEFORE_RREADY	The value of RUSER has changed from its initial value between the time RVALID was asserted, and before RREADY was asserted.	A3.2.1
AXI3-60107	AXI_RUSER_UNKN	RUSER has an X or Z value.	A2.6
AXI3-60108	AXI_RVALID_DEASSERTED_BEFORE_RREADY	RVALID has been de-asserted before RREADY was asserted.	A3.2.1
AXI3-60109	AXI_RVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A slave interface must begin driving RVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI3-60110	AXI_RVALID_UNKN	RVALID has an X or Z value.	A2.6
AXI3-60111	AXI_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_READ	Exclusive read accesses must have address aligned to the total number of bytes in the transaction.	A7.2.4
AXI3-60112	AXI_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_WRITE	Exclusive write accesses must have address aligned to the total number of bytes in the transaction.	A7.2.4
AXI3-60113	AXI_UNALIGNED_ADDR_FOR_WRAPPING_READ_BURST	Wrapping bursts must have address aligned to the start of the read transfer.	A3.4.1
AXI3-60114	AXI_UNALIGNED_ADDR_FOR_WRAPPING_WRITE_BURST	Wrapping bursts must have address aligned to the start of the write transfer.	A3.4.1

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60115	AXI_WDATA_CHANGED_BEFORE_WREADY_ON_INVALID_LANE	On a lane whose strobe is 0, the value of WDATA has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	A3.2.1
AXI3-60116	AXI_WDATA_CHANGED_BEFORE_WREADY_ON_VALID_LANE	On a lane whose strobe is 1, the value of WDATA has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	A3.2.1
AXI3-60117	AXI_WLAST_CHANGED_BEFORE_WREADY	The value of WLAST has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	A3.2.1
AXI3-60118	AXI_WID_CHANGED_BEFORE_WREADY	The value of WID has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	A3.2.1
AXI3-60119	AXI_WLAST_UNKN	WLAST has an X or Z value.	A2.3
AXI3-60120	AXI_WID_UNKN	WID has an X or Z value.	A2.3
AXI3-60121	AXI_WREADY_UNKN	WREADY has an X or Z value.	A2.3
AXI3-60122	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_12	The WA bit of the cache transaction field should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60123	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_13	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60124	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_4	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60125	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_5	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60126	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_8	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60127	AXI_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_9	The WA of the cache transaction field bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI3-60128	AXI_WRITE_BURST_SIZE_VIOLATION	In this write transaction, size has been set greater than the defined data bus width.	A3.4.1

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60129	AXI_WRITE_DATA_BEFORE_ADDRESS	A write data beat has occurred before the corresponding address phase.	-
AXI3-60130	AXI_WRITE_DATA_UNKN_ON_INVALID_LANE	On a lane whose strobe is 0, WDATA has an X or Z value.	A2.3
AXI3-60131	AXI_WRITE_DATA_UNKN_ON_VALID_LANE	On a lane whose strobe is 1, WDATA has an X or Z value.	A2.3
AXI3-60132	AXI_RESERVED_AWLOCK_ENCODING	The reserved encoding of 2'b11 should not be used for AWLOCK.	A7.4
AXI3-60133	AXI_WRITE_STROBE_ON_INVALID_BYTE_LANES	Write strobe(s) incorrect for the address/size of a fixed transfer.	A2.3
AXI3-60134	AXI_WSTRB_CHANGED_BEFORE_WREADY	The value of WSTRB has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	A3.2.1
AXI3-60135	AXI_WSTRB_UNKN	WSTRB has an X or Z value.	A2.3
AXI3-60136	AXI_WUSER_CHANGED_BEFORE_WREADY	The value of WUSER has changed from its initial value between the time WVALID was asserted, and before WREADY was asserted.	A3.2.1
AXI3-60137	AXI_WUSER_UNKN	WUSER has an X or Z value.	A2.3
AXI3-60138	AXI_WVALID_DEASSERTED_BEFORE_WREADY	WVALID has been de-asserted before WREADY was asserted.	A3.2.1
AXI3-60139	AXI_WVALID_HIGH_ON_FIRST_CLOCK_AFTER_RESET	A master interface must begin driving WVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI3-60140	AXI_WVALID_UNKN	WVALID has an X or Z value.	A2.3
AXI3-60141	AXI_ADDR_ACROSS_4K_WITHIN_LOCKED_WRITE_TRANSACTION	Transactions in a locked write sequence should be in the same 4K address boundary.	A7.3
AXI3-60142	AXI_ADDR_ACROSS_4K_WITHIN_LOCKED_READ_TRANSACTION	Transactions in a locked read sequence should be in the same 4K address boundary.	A7.3
AXI3-60143	AXI_AWID_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the AWID signal in the locked transaction.	A7.3
AXI3-60144	AXI_ARID_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the ARID signal in the locked transaction.	A7.3

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60145	AXI_AWPROT_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the AWPROT signal in the locked transaction.	A7.3
AXI3-60146	AXI_ARPROT_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the ARPROT signal in the locked transaction.	A7.3
AXI3-60147	AXI_AWCACHE_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the AWCACHE signal in the locked transaction.	A7.3
AXI3-60148	AXI_ARCACHE_CHANGED_WITHIN_LOCKED_TRANSACTION	Master should not change the ARCACHE signal in the locked transaction.	A7.3
AXI3-60149	AXI_NUMBER_OF_LOCKED_SEQUENCES_EXCEEDS_2	Number of accesses in a locked sequence should not be more than 2.	A7.3
AXI3-60150	AXI_LOCKED_WRITE_BEFORE_COMPLETION_OF_PREVIOUS_WRITE_TRANSACTIONS	A locked write sequence should not commence before completion of all previously issued write addresses.	A7.3
AXI3-60151	AXI_LOCKED_WRITE_BEFORE_COMPLETION_OF_PREVIOUS_READ_TRANSACTIONS	A locked write sequence should not commence before completion of all previously issued read addresses.	A7.3
AXI3-60152	AXI_LOCKED_READ_BEFORE_COMPLETION_OF_PREVIOUS_WRITE_TRANSACTIONS	A locked read sequence should not commence before completion of all previously issued write addresses.	A7.3
AXI3-60153	AXI_LOCKED_READ_BEFORE_COMPLETION_OF_PREVIOUS_READ_TRANSACTIONS	A locked read sequence should not commence before completion of all previously issued read addresses.	A7.3
AXI3-60154	AXI_NEW_BURST_BEFORE_COMPLETION_OF_UNLOCK_TRANSACTION	The unlocking transaction should be completed before further any transactions are initiated.	A7.3
AXI3-60155	AXI_UNLOCKED_WRITE_WHILE_OUTSTANDING_LOCKED_WRITES	Unlocking write transaction started while outstanding locked write transaction has not completed.	A7.3
AXI3-60156	AXI_UNLOCKED_WRITE_WHILE_OUTSTANDING_LOCKED_READS	Unlocking write transaction started while outstanding locked read transaction has not completed.	A7.3
AXI3-60157	AXI_UNLOCKED_READ_WHILE_OUTSTANDING_LOCKED_WRITES	Unlocking read transaction started while outstanding locked write transaction has not completed.	A7.3

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60158	AXI_UNLOCKED_READ_WHILE_OUTSTANDING_LOCKED_READS	Unlocking read transaction started while outstanding locked read transaction has not completed.	A7.3
AXI3-60159	AXI_UNLOCKING_TRANSACTION_WITH_AN_EXCLUSIVE_ACCESS	Unlocking transaction cannot be an exclusive access transaction.	A7.3
AXI3-60160	AXI_FIRST_DATA_ITEM_OF_TRANSACTION_WRITE_ORDER_VIOLATION	The order in which a slave receives the first data item of each transaction must be the same as the order in which it receives the addresses for the transaction.	A5.3.3
AXI3-60161	AXI_AWLEN_MISMATCHED_WITH_COMPLETED_WRITE_DATA_BURST	Actual length of data burst has exceeded the burst length specified by AWLEN.	A3.4.1
AXI3-60162	AXI_WRITE_LENGTH_MISMATCHED_ACTUAL_LENGTH_OF_WRITE_DATA_BURST_EXCEEDS_AWLEN	AWLEN value of write address control does not match with corresponding outstanding write data burst length.	A3.4.1
AXI3-60163	AXI_AWLEN_MISMATCHED_ACTUAL_LENGTH_OF_WRITE_DATA_BURST_EXCEEDS_AWLEN	The actual length of write data burst exceeds with the length specified by AWLEN.	A3.4.1
AXI3-60164	AXI_WLAST_ASSERTED_DURING_DATA_PHASE_OTHER_THAN_LAST	WLAST must only be asserted during the last data phase.	A3.4.1
AXI3-60165	AXI_WRITE_INTERLEAVE_DEPTH_VIOLATION	Write data bursts should not be interleaved beyond the write interleaving depth.	A5.3.3
AXI3-60166	AXI_WRITE_RESPONSE_WITHOUT_ADDR	Write response should not be sent before the corresponding address has completed.	A3.3.1
AXI3-60167	AXI_WRITE_RESPONSE_WITHOUT_DATA	Write response should not be sent before the corresponding write data burst has completed.	A3.3.1
AXI3-60168	AXI_AWVALID_HIGH_DURING_RESET	AWVALID asserted during the reset state.	A3.1.2
AXI3-60169	AXI_WVALID_HIGH_DURING_RESET	WVALID asserted during the reset state	A3.1.2
AXI3-60170	AXI_BVALID_HIGH_DURING_RESET	BVALID asserted during the reset state	A3.1.2
AXI3-60171	AXI_ARVALID_HIGH_DURING_RESET	ARVALID asserted during the reset state	A3.1.2

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60172	AXI_RVALID_HIGH_DURING_RESET	RVALID asserted during the reset state	A3.1.2
AXI3-60173	AXI_RLAST_VIOLATION	RLAST signal should be asserted along with the final transfer of the read data burst.	A3.4.1
AXI3-60174	AXI_EX_WRITE_AFTER_EX_READ_FAILURE	It is recommended that an exclusive write access should not be performed after the corresponding exclusive read failure.	A7.2.2
AXI3-60175	AXI_TIMEOUT_WAITING_FOR_WRITE_DATA	Timed-out waiting for a data phase in write data burst.	A2.3
AXI3-60176	AXI_TIMEOUT_WAITING_FOR_WRITE_RESPONSE	Timed-out waiting for a write response.	A2.4
AXI3-60177	AXI_TIMEOUT_WAITING_FOR_READ_RESPONSE	Timed-out waiting for a read response.	A2.6
AXI3-60178	AXI_TIMEOUT_WAITING_FOR_WRITE_ADDR_AFTER_DATA	Timed-out waiting for a write address phase to be coming after data.	A2.2
AXI3-60179	AXI_DEC_ERR_RESP_FOR_READ	No slave at the address for this read transfer (signaled by <AXI_DECERR>).	
AXI3-60180	AXI_DEC_ERR_RESP_FOR_WRITE	No slave at the address for this write transfer (signaled by <AXI_DECERR>).	
AXI3-60181	AXI_SLV_ERR_RESP_FOR_READ	Slave has detected an error for this read transfer (signaled by <AXI_SLVERR>).	
AXI3-60182	AXI_SLV_ERR_RESP_FOR_WRITE	Slave has detected an error for this write transfer (signaled by <AXI_SLVERR>).	
AXI3-60183	AXI_MINIMUM_SLAVE_ADDRESS_SPACE_VIOLATION	The minimum address space occupied by a single slave device is 4 kilobytes.	A10.3.2
AXI3-60184	AXI_ADDRESS_WIDTH_EXCEEDS_64	AXI supports up to 64-bit addressing.	A10.3.1
AXI3-60185	AXI_READ_BURST_MAXIMUM_LENGTH_VIOLATION	16 read data beats were seen without RLAST.	A3.4.1
AXI3-60186	AXI_WRITE_BURST_MAXIMUM_LENGTH_VIOLATION	16 write data beats were seen without WLAST.	A3.4.1
AXI3-60187	AXI_WRITE_STROBES_LENGTH_VIOLATION	The size of the *write_strobes* array in a write transfer should match the value given by AWLEN.	

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60188	AXI_EX_RD_WHEN_EX_NOT_ENABLED	An exclusive read should not be issued when exclusive transactions are not enabled.	
AXI3-60189	AXI_EX_WR_WHEN_EX_NOT_ENABLED	An exclusive write should not be issued when exclusive transactions are not enabled.	
AXI3-60190	AXI_WRITE_TRANSFER_EXCEEDS_ADDRESS_SPACE	This write transfer runs off the edge of the address space defined by AXI_ADDRESS_WIDTH.	A10.3.1
AXI3-60191	AXI_READ_TRANSFER_EXCEEDS_ADDRESS_SPACE	This read transfer runs off the edge of the address space defined by AXI_ADDRESS_WIDTH.	A10.3.1
AXI3-60192	AXI_EXCL_RD_WHILE_EXCL_WR_IN_PROGRESS_SAME_ID	Master starts an exclusive read burst while exclusive write burst with same ID tag is in progress.	A7.2.4
AXI3-60193	AXI_EXCL_WR_WHILE_EXCL_RD_IN_PROGRESS_SAME_ID	Master starts an exclusive write burst while exclusive read burst with same ID tag is in progress.	A7.2.4
AXI3-60194	AXI_ILLEGAL_LENGTH_READ_BURST	Read address phase *burst_length* has an illegal value.	A3.4.1
AXI3-60195	AXI_ILLEGAL_LENGTH_WRITE_BURST	Write address phase burst_length has an illegal value.	A3.4.1
AXI3-60196	AXI_ARREADY_NOT_ASSERTED_AFTER_ARVALID	When ARVALID has been asserted, ARREADY should be asserted within *config_max_latency_ARVALID_assertion_to_ARREADY* clock periods.	
AXI3-60197	AXI_BREADY_NOT_ASSERTED_AFTER_BVALID	When BVALID has been asserted, *BREADY* should be asserted within *config_max_latency_BVALID_assertion_to_BREADY* clock periods.	
AXI3-60198	AXI_AWREADY_NOT_ASSERTED_AFTER_AWVALID	When AWVALID has been asserted, AWREADY should be asserted within *config_max_latency_AWVALID_assertion_to_AWREADY* clock periods.	
AXI3-60199	AXI_RREADY_NOT_ASSERTED_AFTER_RVALID	When RVALID has been asserted, RREADY should be asserted within *config_max_latency_RVALID_assertion_to_RREAD*Y clock periods.	
AXI3-60200	AXI_WREADY_NOT_ASSERTED_AFTER_WVALID	When WVALID has been asserted, WREADY should be asserted within *config_max_latency_WVALID_assertion_to_WREADY* clock periods.	
AXI3-60201	AXI_DEC_ERR_ILLEGAL_FOR_MAPPED_SLAVE_ADDR	Slave receives a burst to a mapped address but responds with DECERR (signaled by AXI_DECERR).	A3.4.4

Table A-1. AXI3 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI3-60202	AXI_PARAM_READ_REORDERING_DEPTH_EQUALS_ZERO	The user-supplied *config_read_data_reordering_depth* should be greater than zero.	A5.3.1
AXI3-60203	AXI_PARAM_READ_REORDERING_DEPTH_EXCEEDS_MAX_ID	The user-supplied *config_read_data_reordering_depth* exceeds the maximum possible value, as defined by the AXI_ID_WIDTH parameter.	A5.3.1
AXI3-60204	AXI_READ_REORDERING_VIOLATION	The arrival of a read response has exceeded the read reordering depth.	A5.3.1
AXI3-60205	AXI_READ_ISSUING_CAPABILITY_VIOLATION	The number of outstanding Read transactions exceeded the maximum Read issuing capability.	
AXI3-60206	AXI_WRITE_ISSUING_CAPABILITY_VIOLATION	The number of outstanding Write transactions exceeded the maximum Write issuing capability.	
AXI3-60207	AXI_COMBINED_ISSUING_CAPABILITY_VIOLATION	The number of outstanding Read and Write transactions exceeded the maximum combined issuing capability.	
AXI3-60208	AXI_READ_ACCEPTANCE_CAPABILITY_VIOLATION	The number of outstanding Read transactions exceeded the maximum Read acceptance capability.	
AXI3-60209	AXI_WRITE_ACCEPTANCE_CAPABILITY_VIOLATION	The number of outstanding Write transactions exceeded the maximum Write acceptance capability.	
AXI3-60210	AXI_COMBINED_ACCEPTANCE_CAPABILITY_VIOLATION	The number of outstanding Read and Write transactions exceeded the maximum combined acceptance capability.	

AXI4 Assertions

The AXI4 master, slave, and monitor BFMs all support error checking with the firing of one or more assertions when a property defined in the AMBA AXI Protocol Specification has been violated. Each assertion can be individually enabled/disabled using the *set_config()* function for a particular BFM. The property covered for each assertion is noted in Table A-2 under the Property Reference column. The reference number refers to the section number in the AMBA AXI Protocol Specification.

Table A-2. AXI4 Assertions

Error Code	Error Name	Description	Property Ref
AXI4-60000	AXI4_ADDRESS_WIDTH_EXCEEDS_64	AXI4 supports up to 64-bit addressing.	A10.3.1
AXI4-60001	AXI4_ADDR_FOR_READ_BURST_ACROSS_4K_BOUNDARY	This read transaction has crossed a 4KB boundary.	A3.4.1
AXI4-60002	AXI4_ADDR_FOR_WRITE_BURST_ACROSS_4K_BOUNDARY	This write transaction has crossed a 4KB boundary.	A3.4.1
AXI4-60003	AXI4_ARADDR_CHANGED_BEFORE_ARREADY	The value of ARADDR has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60004	AXI4_ARADDR_FALLS_IN_REGION_HOLE	The ARADDR value cannot be decoded to a region in the region map.	A8.2.1
AXI4-60005	AXI4_ARADDR_UNKN	ARADDR has an X value/ARADDR has a Z value.	
AXI4-60006	AXI4_ARBURST_CHANGED_BEFORE_ARREADY	The value of ARBURST has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60007	AXI4_ARBURST_UNKN	ARBURST has an X value/ARBURST has a Z value.	
AXI4-60008	AXI4_ARCACHE_CHANGED_BEFORE_ARREADY	The value of ARCACHE has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60009	AXI4_ARCACHE_UNKN	ARCACHE has an X value/ARCACHE has a Z value.	
AXI4-60010	AXI4_ARID_CHANGED_BEFORE_ARREADY	The value of ARID has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60011	AXI4_ARID_UNKN	ARID has an X value/ARID has a Z value.	
AXI4-60012	AXI4_ARLEN_CHANGED_BEFORE_ARREADY	The value of ARLEN has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60013	AXI4_ARLEN_UNKN	ARLEN has an X value/ARLEN has a Z value.	

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60014	AXI4_ARLOCK_CHANGED_BEFORE_ARREADY	The value of ARLOCK has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60015	AXI4_ARLOCK_UNKN	ARLOCK has an X value/ARLOCK has a Z value.	
AXI4-60016	AXI4_ARPROT_CHANGED_BEFORE_ARREADY	The value of ARPROT has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60017	AXI4_ARPROT_UNKN	ARPROT has an X value/ARPROT has a Z value.	
AXI4-60018	AXI4_ARQOS_CHANGED_BEFORE_ARREADY	The value of ARQOS has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60019	AXI4_ARQOS_UNKN	ARQOS has an X value/ARQOS has a Z value.	
AXI4-60020	AXI4_ARREADY_NOT_ASSERTED_AFTER_ARVALID	Once ARVALID has been asserted ARREADY should be asserted in config_max_latency_ARVALID_assertion_to_ARREADY clock periods.	
AXI4-60021	AXI4_ARREADY_UNKN	ARREADY has an X value/ARREADY has a Z value.	
AXI4-60022	AXI4_ARREGION_CHANGED_BEFORE_ARREADY	The value of ARREGION has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60023	AXI4_ARREGION_MISMATCH	The ARREGION value does not match the value defined in the region map.	A8.2.1
AXI4-60024	AXI4_ARREGION_UNKN	ARREGION has an X value/ARREGION has a Z value.	
AXI4-60025	AXI4_ARSIZE_CHANGED_BEFORE_ARREADY	The value of ARSIZE has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60026	AXI4_ARSIZE_UNKN	ARSIZE has an X value/ARSIZE has a Z value.	
AXI4-60027	AXI4_ARUSER_CHANGED_BEFORE_ARREADY	The value of ARUSER has changed from its initial value between the time ARVALID was asserted and before ARREADY was asserted.	A3.2.1
AXI4-60028	AXI4_ARUSER_UNKN	ARUSER has an X value/ARUSER has a Z value.	

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60029	AXI4_ARVALID_DEASSERTED_BEFORE_ARREADY	ARVALID has been de-asserted before ARREADY was asserted.	A3.2.1
AXI4-60030	AXI4_ARVALID_HIGH_ON_FIRST_CLOCK	A master interface must begin driving ARVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI4-60031	AXI4_ARVALID_UNKN	ARVALID has an X value/ARVALID has a Z value.	
AXI4-60032	AXI4_AWADDR_CHANGED_BEFORE_AWREADY	The value of AWADDR has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60033	AXI4_AWADDR_FALLS_IN_REGION_HOLE	The addr value cannot be decoded to a region in the region map.	A8.2.1
AXI4-60034	AXI4_AWADDR_UNKN	AWADDR has an X value/AWADDR has a Z value.	
AXI4-60035	AXI4_AWBURST_CHANGED_BEFORE_AWREADY	The value of AWBURST has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60036	AXI4_AWBURST_UNKN	AWBURST has an X value/AWBURST has a Z value.	
AXI4-60037	AXI4_AWCACHE_CHANGED_BEFORE_AWREADY	The value of AWCACHE has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60038	AXI4_AWCACHE_UNKN	AWCACHE has an X value/AWCACHE has a Z value.	
AXI4-60039	AXI4_AWID_CHANGED_BEFORE_AWREADY	The value of AWID has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60040	AXI4_AWID_UNKN	AWID has an X value/AWID has a Z value.	
AXI4-60041	AXI4_AWLEN_CHANGED_BEFORE_AWREADY	The value of AWLEN has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60042	AXI4_AWLEN_UNKN	AWLEN has an X value/AWLEN has a Z value.	
AXI4-60043	AXI4_AWLOCK_CHANGED_BEFORE_AWREADY	The value of AWLOCK has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60044	AXI4_AWLOCK_UNKN	AWLOCK has an X value/AWLOCK has a Z value.	
AXI4-60045	AXI4_AWPROT_CHANGED_BEFORE_AWREADY	The value of AWPROT has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60046	AXI4_AWPROT_UNKN	AWPROT has an X value/AWPROT has a Z value.	
AXI4-60047	AXI4_AWQOS_CHANGED_BEFORE_AWREADY	The value of AWQOS has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60048	AXI4_AWQOS_UNKN	AWQOS has an X value/AWQOS has a Z value.	
AXI4-60049	AXI4_AWREADY_NOT_ASSERTED_AFTER_AWVALID	Once AWVALID has been asserted AWREADY should be asserted in *config_max_latency_AWVALID_assertion_to_AWREADY* clock periods.	
AXI4-60050	AXI4_AWREADY_UNKN	AWREADY has an X value/AWREADY has a Z value.	
AXI4-60051	AXI4_AWREGION_CHANGED_BEFORE_AWREADY	The value of AWREGION has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60052	AXI4_AWREGION_MISMATCH	The AWREGION value does not match the value defined in the region map.	A8.2.1
AXI4-60053	AXI4_AWREGION_UNKN	AWREGION has an X value/AWREGION has a Z value.	
AXI4-60054	AXI4_AWSIZE_CHANGED_BEFORE_AWREADY	The value of AWSIZE has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60055	AXI4_AWSIZE_UNKN	AWSIZE has an X value/AWSIZE has a Z value.	
AXI4-60056	AXI4_AWUSER_CHANGED_BEFORE_AWREADY	The value of AWUSER has changed from its initial value between the time AWVALID was asserted and before AWREADY was asserted.	A3.2.1
AXI4-60057	AXI4_AWUSER_UNKN	AWUSER has an X value/AWUSER has a Z value.	
AXI4-60058	AXI4_AWVALID_DEASSERTED_BEFORE_AWREADY	AWVALID has been de-asserted before AWREADY was asserted.	A3.2.1

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60059	AXI4_AWVALID_HIGH_ON_FIRST_CLOCK	A master interface must begin driving AWVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI4-60060	AXI4_AWVALID_UNKN	AWVALID has an X value/AWVALID has a Z value.	
AXI4-60061	AXI4_BID_CHANGED_BEFORE_BREADY	The value of BID has changed from its initial value between the time BVALID was asserted and before BREADY was asserted.	A3.2.1
AXI4-60062	AXI4_BID_UNKN	BID has an X value/BID has a Z value.	
AXI4-60063	AXI4_BREADY_NOT_ASSERTED_AFTER_BVALID	When BVALID has been asserted BREADY should be asserted in c*onfig_max_latency_BVALID_assertion_to_BREADY* clock periods.	
AXI4-60064	AXI4_BREADY_UNKN	BREADY has an X value/BREADY has a Z value.	
AXI4-60065	AXI4_BRESP_CHANGED_BEFORE_BREADY	The value of BRESP has changed from its initial value between the time BVALID was asserted and before BREADY was asserted.	A3.2.1
AXI4-60066	AXI4_BRESP_UNKN	BRESP has an X value/BRESP has a Z value.	
AXI4-60067	AXI4_BUSER_CHANGED_BEFORE_BREADY	The value of BUSER has changed from its initial value between the time BVALID was asserted and before BREADY was asserted.	A3.2.1
AXI4-60068	AXI4_BUSER_UNKN	BUSER has an X value/BUSER has a Z value.	
AXI4-60069	AXI4_BVALID_DEASSERTED_BEFORE_BREADY	BVALID has been de-asserted before BREADY was asserted.	A3.2.1
AXI4-60070	AXI4_BVALID_HIGH_EXITING_RESET	BVALID should have been driven low when exiting reset.	A3.1.2
AXI4-60071	AXI4_BVALID_UNKN	BVALID has an X value/BVALID has a Z value.	
AXI4-60072	AXI4_DEC_ERR_RESP_FOR_READ	No slave at the address for this read transfer (signaled by AXI4_DECERR).	
AXI4-60073	AXI4_DEC_ERR_RESP_FOR_WRITE	No slave at the address for this write transfer (signaled by AXI4_DECERR).	
AXI4-60074	AXI4_EXCLUSIVE_READ_ACCESS_MODIFIABLE	The modifiable bit (bit 1 of the cache parameter) should not be set for an exclusive read access.	A7.2.4

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60075	AXI4_EXCLUSIVE_READ_BYTES_TRANSFER_EXCEEDS_128	Number of bytes in an exclusive read transaction must be less than or equal to 128.	A7.2.4
AXI4-60076	AXI4_EXCLUSIVE_READ_BYTES_TRANSFER_NOT_POWER_OF_2	Number of bytes of an exclusive read transaction is not a power of 2.	A7.2.4
AXI4-60077	AXI4_EXCLUSIVE_READ_LENGTH_EXCEEDS_16	Exclusive read accesses are not permitted to use a burst length greater than 16.	A7.2.4
AXI4-60078	AXI4_EXCLUSIVE_WR_ADDRESS_NOT_SAME_AS_RD	Exclusive write does not match the address of the previous exclusive read to this id.	A7.2.4
AXI4-60079	AXI4_EXCLUSIVE_WR_BURST_NOT_SAME_AS_RD	Exclusive write does not match the burst setting of the previous exclusive read to this id.	A7.2.4
AXI4-60080	AXI4_EXCLUSIVE_WR_CACHE_NOT_SAME_AS_RD	Exclusive write does not match the cache setting of the previous exclusive read to this id (see the *ARM AXI4 compliance-checker AXI4_RECM_EXCL_MATCH* assertion code).	
AXI4-60081	AXI4_EXCLUSIVE_WRITE_ACCESS_MODIFIABLE	The modifiable bit (bit 1 of the cache parameter) should not be set for an exclusive write access.	A7.2.4
AXI4-60082	AXI4_EXCLUSIVE_WR_LENGTH_NOT_SAME_AS_RD	Exclusive write does not match the length of the previous exclusive read to this id.	A7.2.4
AXI4-60083	AXI4_EXCLUSIVE_WR_PROT_NOT_SAME_AS_RD	Exclusive write does not match the prot setting of the previous exclusive read to this id.	A7.2.4
AXI4-60084	AXI4_EXCLUSIVE_WR_REGION_NOT_SAME_AS_RD	Exclusive write does not match the region setting of the previous exclusive read to this id.	A7.2.4
AXI4-60085	AXI4_EXCLUSIVE_WR_SIZE_NOT_SAME_AS_RD	Exclusive write does not match the size of the previous exclusive read to this id.	A7.2.4
AXI4-60086	AXI4_EXOKAY_RESPONSE_NORMAL_READ	Slave has responded AXI4_EXOKAY to a non exclusive read transfer.	
AXI4-60087	AXI4_EXOKAY_RESPONSE_NORMAL_WRITE	Slave has responded AXI4_EXOKAY to a non exclusive write transfer.	
AXI4-60088	AXI4_EX_RD_EXOKAY_RESP_EXPECTED_OKAY	Expected AXI4_OKAY response to this exclusive read (because the parameters did not meet the the restrictions) but got AXI4_EXOKAY.	A7.2.4

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60089	AXI4_EX_RD_EXOKAY_RESP_SLAVE_WITHOUT_EXCLUSIVE_ACCESS	Response for an exclusive read to a slave which does not support exclusive access should be AXI4_OKAY but it returned AXI4_EXOKAY.	A7.2.5
AXI4-60090	AXI4_EX_RD_OKAY_RESP_EXPECTED_EXOKAY	Expected AXI4_EXOKAY response to this exclusive read (because the parameters met the restrictions) but got AXI4_OKAY.	A7.2.4
AXI4-60091	AXI4_EX_RD_WHEN_EX_NOT_ENABLED	An exclusive read should not be issued when exclusive transactions are not enabled.	
AXI4-60092	AXI4_EX_WRITE_BEFORE_EX_READ_RESPONSE	Exclusive write has occurred with no previous exclusive read.	
AXI4-60093	AXI4_EX_WRITE_EXOKAY_RESP_EXPECTED_OKAY	Exclusive write has not been successful but slave has responded with AXI4_EXOKAY.	A7.2.2
AXI4-60094	AXI4_EX_WRITE_EXOKAY_RESP_SLAVE_WITHOUT_EXCLUSIVE_ACCESS	Response for an exclusive write to a slave which does not support exclusive access should be AXI4_OKAY but it returned AXI4_EXOKAY.	A7.2.5
AXI4-60095	AXI4_EX_WRITE_OKAY_RESP_EXPECTED_EXOKAY	An AXI4_OKAY response to an exclusive write occurred but an AXI4_EXOKAY response had been expected. If the slave has multiple interfaces to the system this check should be disabled as it is possible for this response to occur as a result of activity on another port.	A7.2.2
AXI4-60096	AXI4_EX_WR_WHEN_EX_NOT_ENABLED	An exclusive write should not be issued when exclusive transactions are not enabled.	
AXI4-60097	AXI4_ILLEGAL_ARCACHE_VALUE_FOR_CACHEABLE_ADDRESS_REGION	For a read from a cacheable address region one of bits 2 or 3 of the cache parameter must be HIGH.	A4.5
AXI4-60098	AXI4_ILLEGAL_ARCACHE_VALUE_FOR_NON_CACHEABLE_ADDRESS_REGION	For a read from a non-cacheable address region bits 2 and 3 of the cache parameter must be LOW.	A4.5
AXI4-60099	AXI4_ILLEGAL_AWCACHE_VALUE_FOR_CACHEABLE_ADDRESS_REGION	For a write to a cacheable address region one of bits 2 or 3 of the cache parameter must be HIGH.	A4.5
AXI4-60100	AXI4_ILLEGAL_AWCACHE_VALUE_FOR_NON_CACHEABLE_ADDRESS_REGION	For a write to a non-cacheable address region bits 2 and 3 of the cache parameter must be LOW.	A4.5
AXI4-60101	AXI4_ILLEGAL_LENGTH_FIXED_READ_BURST	In the last read address phase burst_length has an illegal value for a burst of type AXI4_FIXED.	A3.4.1

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60102	AXI4_ILLEGAL_LENGTH_FIXED_WRITE_BURST	In the last write address phase burst_length has an illegal value for a burst of type AXI4_FIXED.	A3.4.1
AXI4-60103	AXI4_ILLEGAL_LENGTH_WRAPPING_READ_BURST	In the last read address phase burst_length has an illegal value for a burst of type AXI4_WRAP.	A3.4.1
AXI4-60104	AXI4_ILLEGAL_LENGTH_WRAPPING_WRITE_BURST	In the last write address phase burst_length has an illegal value for a burst of type AXI4_WRAP.	A3.4.1
AXI4-60105	AXI4_ILLEGAL_RESPONSE_EXCLUSIVE_READ	Response for an exclusive read should be either AXI4_OKAY or AXI4_EXOKAY.	
AXI4-60106	AXI4_ILLEGAL_RESPONSE_EXCLUSIVE_WRITE	Response for an exclusive write should be either AXI4_OKAY or AXI4_EXOKAY.	
AXI4-60107	AXI4_INVALID_REGION_CARDINALITY	The configuration parameter config_slave_regions does not lie in the range 1-16 inclusive.	A8.2.1.
AXI4-60108	AXI4_INVALID_WRITE_STROBES_ON_ALIGNED_WRITE_TRANSFER	Write strobe(s) incorrect for address/size of an aligned transaction.	A3.4.3
AXI4-60109	AXI4_INVALID_WRITE_STROBES_ON_UNALIGNED_WRITE_TRANSFER	Write strobe(s) incorrect for address/size of an unaligned transaction.	A3.4.3
AXI4-60110	AXI4_MINIMUM_SLAVE_ADDRESS_SPACE_VIOLATION	The minimum address space occupied by a single slave device is 4 kilobytes.	A10.3.2
AXI4-60111	AXI4_NON_INCREASING_REGION_SPECIFICATION	A region address-range has an upper bound smaller than the lower bound.	
AXI4-60112	AXI4_NON_ZERO_ARQOS	The master is configured to not participate in the Quality-of-Service scheme but ARQOS is not 4'b0000 as it should be.	A8.1.2
AXI4-60113	AXI4_NON_ZERO_AWQOS	The master is configured to not participate in the Quality-of-Service scheme but AWQOS is not 4'b0000 as it should be.	A8.1.2
AXI4-60114	AXI4_OVERLAPPING_REGION	An address-range in the region map overlaps with another address in the region map.	A8.2.1.
AXI4-60115	AXI4_PARAM_READ_DATA_BUS_WIDTH	The value of AXI4_RDATA_WIDTH must be one of 8,16,32,64,128,256,512, or 1024.	A1.3.1

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60116	AXI4_PARAM_READ_REORDERING_DEPTH_EQUALS_ZERO	The user-supplied *config_read_data_reordering_depth* should be greater than zero.	A5.3.1
AXI4-60117	AXI4_PARAM_READ_REORDERING_DEPTH_EXCEEDS_MAX_ID	The user-supplied *config_read_data_reordering_depth* exceeds the maximum possible value as defined by the AXI4_ID_WIDTH parameter.	A5.3.1
AXI4-60118	AXI4_PARAM_WRITE_DATA_BUS_WIDTH	The value of AXI4_WDATA_WIDTH must be one of 8,16,32,64,128,256,512, or 1024.	A1.3.1
AXI4-60119	AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_12	The RA bit of the cache parameter should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60120	AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_13	The RA bit of the cache parameter should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60121	AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_4	The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60122	AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_5	The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60123	AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_8	The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60124	AXI4_READ_ALLOCATE_WHEN_NON_MODIFIABLE_9	The RA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60125	AXI4_READ_BURST_LENGTH_VIOLATION	The *burst_length* implied by the number of beats actually read does not match the *burst_length* defined by the *master_read_addr_channel_phase*.	
AXI4-60126	AXI4_READ_BURST_MAXIMUM_LENGTH_VIOLATION	256 read data beats were seen without RLAST.	A3.4.1
AXI4-60127	AXI4_READ_BURST_SIZE_VIOLATION	In this read transaction, size has been set too high for the defined data buswidth.	
AXI4-60128	AXI4_READ_DATA_BEFORE_ADDRESS	An unexpected read response has occurred (there are no outstanding read transactions with this id).	A3.3.1
AXI4-60129	AXI4_READ_DATA_CHANGED_BEFORE_RREADY	The value of RDATA has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.	A3.2.1

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60130	AXI4_READ_DATA_UNKN	RDATA has an X value/RDATA has a Z value.	
AXI4-60131	AXI4_READ_EXCLUSIVE_ENCODING_VIOLATION.	A read-only interface does not support exclusive accesses.	A10.2.2
AXI4-60132	AXI4_READ_REORDERING_VIOLATION	The arrival of a read response has exceeded the read reordering depth.	A5.3.1
AXI4-60133	AXI4_READ_RESP_CHANGED_BEFORE_RREADY	The value of RRESP has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.	A3.2.1
AXI4-60134	AXI4_READ_TRANSFER_EXCEEDS_ADDRESS_SPACE	This read transfer runs off the edge of the address space defined by AXI4_ADDRESS_WIDTH.	A10.3.1
AXI4-60135	AXI4_REGION_SMALLER_THAN_4KB	An address-range in the region map is smaller than 4kB.	A8.2.1
AXI4-60136	AXI4_RESERVED_ARBURST_ENCODING	The reserved encoding of 2'b11 should not be used for ARBURST.	A3.4.1
AXI4-60137	AXI4_RESERVED_AWBURST_ENCODING	The reserved encoding of 2'b11 should not be used for AWBURST.	A3.4.1
AXI4-60138	AXI4_RID_CHANGED_BEFORE_RREADY	The value of RID has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.	A3.2.1
AXI4-60139	AXI4_RID_UNKN	RID has an X value/RID has a Z value.	
AXI4-60140	AXI4_RLAST_CHANGED_BEFORE_RREADY	The value of RLAST has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.	A3.2.1
AXI4-60141	AXI4_RLAST_UNKN	RLAST has an X value/RLAST has a Z value.	
AXI4-60142	AXI4_RREADY_NOT_ASSERTED_AFTER_RVALID	Once RVALID has been asserted RREADY should be asserted in *config_max_latency_RVALID_assertion_to_RREADY* clock periods.	
AXI4-60143	AXI4_RREADY_UNKN	RREADY has an X value/RREADY has a Z value.	
AXI4-60144	AXI4_RRESP_UNKN	RRESP has an X value/RRESP has a Z value.	
AXI4-60145	AXI4_RUSER_CHANGED_BEFORE_RREADY	The value of RUSER has changed from its initial value between the time RVALID was asserted and before RREADY was asserted.	A3.2.1
AXI4-60146	AXI4_RUSER_UNKN	RUSER has an X value/RUSER has a Z value.	

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60147	AXI4_RVALID_DEASSERTED_BEFORE_RREADY	RVALID has been de-asserted before RREADY was asserted.	A3.2.1
AXI4-60148	AXI4_RVALID_HIGH_EXITING_RESET	RVALID should have been driven low when exiting reset.	A3.1.2
AXI4-60149	AXI4_RVALID_UNKN	RVALID has an X value/RVALID has a Z value.	
AXI4-60150	AXI4_SLV_ERR_RESP_FOR_READ	Slave has detected an error for this read transfer (signaled by AXI4_SLVERR).	
AXI4-60151	AXI4_SLV_ERR_RESP_FOR_WRITE	Slave has detected an error for this write transfer (signaled by AXI4_SLVERR).	
AXI4-60152	AXI4_TIMEOUT_WAITING_FOR_READ_RESPONSE	Timed-out waiting for a read response.	A4.6
AXI4-60153	AXI4_TIMEOUT_WAITING_FOR_WRITE_RESPONSE	Timed-out waiting for a write response.	A4.6
AXI4-60154	AXI4_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_READ	Exclusive read accesses must have address aligned to the total number of bytes in the transaction.	A7.2.4
AXI4-60155	AXI4_UNALIGNED_ADDR_FOR_WRAPPING_READ_BURST	Wrapping bursts must have address aligned to the start of the read transfer.	A3.4.1
AXI4-60156	AXI4_UNALIGNED_ADDR_FOR_WRAPPING_WRITE_BURST	Wrapping bursts must have address aligned to the start of the write transfer.	A3.4.1
AXI4-60157	AXI4_WDATA_CHANGED_BEFORE_WREADY_ON_INVALID_LANE	On a lane whose strobe is 0, the value of WDATA has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.	A3.2.1
AXI4-60158	AXI4_WDATA_CHANGED_BEFORE_WREADY_ON_VALID_LANE	On a lane whose strobe is 1, the value of WDATA has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.	A3.2.1
AXI4-60159	AXI4_WLAST_CHANGED_BEFORE_WREADY	The value of WLAST has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.	A3.2.1
AXI4-60160	AXI4_WLAST_UNKN	WLAST has an X value/WLAST has a Z value.	
AXI4-60161	AXI4_WREADY_NOT_ASSERTED_AFTER_WVALID	Once WVALID has been asserted WREADY should be asserted in *config_max_latency_WVALID_assertion_to_WREADY* clock periods.	

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60062	AXI4_WREADY_UNKN	WREADY has an X value/WREADY has a Z value.	
AXI4-60163	AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_12	The WA bit of the cache parameter should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60164	AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_13	The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60165	AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_4	The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60166	AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_5	The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60167	AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_8	The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60168	AXI4_WRITE_ALLOCATE_WHEN_NON_MODIFIABLE_9	The WA of the cache parameter bit should not be HIGH when the Modifiable bit is LOW.	A4.4
AXI4-60169	AXI4_WRITE_BURST_LENGTH_VIOLATION	The number of data beats in a write transfer should match the value given by AWLEN.	
AXI4-60170	AXI4_WRITE_STROBES_LENGTH_VIOLATION	The size of the write_strobes array in a write transfer should match the value given by AWLEN.	
AXI4-60171	AXI4_WRITE_USER_DATA_LENGTH_VIOLATION	The size of the wdata_user_data array in a write transfer should match the value given by AWLEN.	
AXI4-60172	AXI4_WRITE_BURST_MAXIMUM_LENGTH_VIOLATION	256 write data beats were seen without WLAST.	A3.4.1
AXI4-60173	AXI4_WRITE_BURST_SIZE_VIOLATION	In this write transaction size has been set too high for the defined data buswidth.	
AXI4-60174	AXI4_WRITE_DATA_BEFORE_ADDRESS	A write data beat has occurred before the corresponding address phase.	
AXI4-60175	AXI4_WRITE_DATA_UNKN_ON_INVALID_LANE	On a lane whose strobe is 0 WDATA has an X value/WDATA has a Z value.	
AXI4-60176	AXI4_WRITE_DATA_UNKN_ON_VALID_LANE	On a lane whose strobe is 1 WDATA has an X value/WDATA has a Z value.	
AXI4-60177	AXI4_WRITE_EXCLUSIVE_ENCODING_VIOLATION	A write-only interface does not support exclusive accesses.	A10.2.3

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60178	AXI4_WRITE_RESPONSE_WITHOUT_ADDR_DATA	An unexpected write response has occurred (there are no outstanding write transactions with this id).	
AXI4-60179	AXI4_WRITE_STROBE_FIXED_BURST_VIOLATION	Write strobe(s) incorrect for the address/size of a fixed transfer.	
AXI4-60180	AXI4_WRITE_TRANSFER_EXCEEDS_ADDRESS_SPACE	This write transfer runs off the edge of the address space defined by AXI4_ADDRESS_WIDTH.	A10.3.1
AXI4-60181	AXI4_WRONG_ARREGION_FOR_SLAVE_WITH_SINGLE_ADDRESS_DECODE	The region value should be 4'b0000 for a read from a slave with a single address decode in the region map.	A8.2.1
AXI4-60182	AXI4_WRONG_AWREGION_FOR_SLAVE_WITH_SINGLE_ADDRESS_DECODE	The region value should be 4'b0000 for a write to a slave with a single address decode in the region map.	A8.2.1
AXI4-60183	AXI4_WSTRB_CHANGED_BEFORE_WREADY	The value of WSTRB has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.	A3.2.1
AXI4-60184	AXI4_WSTRB_UNKN	WSTRB has an X value/WSTRB has a Z value.	
AXI4-60185	AXI4_WUSER_CHANGED_BEFORE_WREADY	The value of WUSER has changed from its initial value between the time WVALID was asserted and before WREADY was asserted.	A3.2.1
AXI4-60186	AXI4_WUSER_UNKN	WUSER has an X value/WUSER has a Z value.	
AXI4-60187	AXI4_WVALID_DEASSERTED_BEFORE_WREADY	WVALID has been de-asserted before WREADY was asserted.	A3.2.1
AXI4-60188	AXI4_WVALID_HIGH_ON_FIRST_CLOCK	A master interface must begin driving WVALID high only at a rising clock edge after ARESETn is HIGH.	A3.1.2
AXI4-60189	AXI4_WVALID_UNKN	WVALID has an X value/WVALID has a Z value.	
AXI4-60190	MVC_FAILED_POSTCONDITION	A postcondition failed.	
AXI4-60191	MVC_FAILED_RECOGNITION	An item failed to be recognized.	
AXI4-60192	AXI4_TIMEOUT_WAITING_FOR_WRITE_DATA	Timed-out waiting for a data phase in write data burst.	A4.6
AXI4-60193	AXI4_EXCL_RD_WHILE_EXCL_WR_IN_PROGRESS_SAME_ID	Master starts an exclusive read burst while exclusive write burst with same ID tag is in progress.	A7.2.4
AXI4-60194	AXI4_EXCL_WR_WHILE_EXCL_RD_IN_PROGRESS_SAME_ID	Master starts an exclusive write burst while exclusive read burst with same ID tag is in progress.	A7.2.4

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60195	AXI4_DEC_ERR_ILLEGAL_FOR_MAPPED_SLAVE_ADDR	Slave receives a burst to a mapped address but responds with DECERR (signaled by AXI4_DECERR).	A3.4.4
AXI4-60196	AXI4_AWVALID_HIGH_DURING_RESET	AWVALID asserted during the reset state.	A3.1.2
AXI4-60197	AXI4_WVALID_HIGH_DURING_RESET	WVALID asserted during the reset state.	A3.1.2
AXI4-60198	AXI4_BVALID_HIGH_DURING_RESET	BVALID asserted during the reset state.	A3.1.2
AXI4-60199	AXI4_ARVALID_HIGH_DURING_RESET	ARVALID asserted during the reset state.	A3.1.2
AXI4-60200	AXI4_RVALID_HIGH_DURING_RESET	RVALID asserted during the reset state.	A3.1.2
AXI4-60201	AXI4_ARESETn_SIGNAL_Z	Reset signal has a Z value.	
AXI4-60202	AXI4_ARESETn_SIGNAL_X	Reset signal has an X value.	
AXI4-60203	AXI4_TIMEOUT_WAITING_FOR_WRITE_ADDR_AFTER_DATA	Timed-out waiting for a write address phase to be coming after data.	A2.2
AXI4-60204	AXI4_EXCLUSIVE_WRITE_BYTES_TRANSFER_EXCEEDS_128	Number of bytes in an exclusive write transaction must be less than or equal to 128.	A7.2.4
AXI4-60205	AXI4_EXCLUSIVE_WRITE_BYTES_TRANSFER_NOT_POWER_OF_2	Number of bytes of an exclusive write transaction is not a power of 2.	A7.2.4
AXI4-60206	AXI4_UNALIGNED_ADDRESS_FOR_EXCLUSIVE_WRITE	Exclusive write accesses must have address aligned to the total number of bytes in the transaction.	A7.2.4
AXI4-60207	AXI4_RLAST_VIOLATION	RLAST signal should be asserted along with the final transfer of the read data burst.	
AXI4-60208	AXI4_WLAST_ASSERTED_DURING_DATA_PHASE_OTHER_THAN_LAST	Wlast must only be asserted during the last data phase.	A3.4.1
AXI4-60209	AXI4_READ_ISSUING_CAPABILITY_VIOLATION	The number of outstanding Read transactions exceeded the maximum Read issuing capability.	
AXI4-60210	AXI4_WRITE_ISSUING_CAPABILITY_VIOLATION	The number of outstanding Write transactions exceeded the maximum Write issuing capability.	
AXI4-60211	AXI4_COMBINED_ISSUING_CAPABILITY_VIOLATION	The number of outstanding Read and Write transactions exceeded the maximum combined issuing capability.	

Table A-2. AXI4 Assertions (cont.)

Error Code	Error Name	Description	Property Ref
AXI4-60212	AXI4_READ_ACCEPTANCE_CAPABILITY_VIOLATION	The number of outstanding Read transactions exceeded the maximum Read acceptance capability.	
AXI4-60213	AXI4_WRITE_ACCEPTANCE_CAPABILITY_VIOLATION	The number of outstanding Write transactions exceeded the maximum Write acceptance capability.	
AXI4-60214	AXI4_COMBINED_ACCEPTANCE_CAPABILITY_VIOLATION	The number of outstanding Read and Write transactions exceeded the maximum combined acceptance capability.	

Third-Party Software for Mentor VIP – Intel FPGA Edition

This section provides information on open source and third-party software that may be included in the Mentor VIP – Intel FPGA Edition software product.

This software application may include GNU GCC version 4.5.0 third-party software. GNU GCC version 4.5.0 is distributed under the terms of the GNU General Public License version 3.0 and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <path to legal directory>/legal/gnu_gpl_3.0.pdf. Portions of this software may be subject to the GNU Free Documentation License version 1.2. You can view a copy of the GNU Free Documentation License version 1.2 at: <path to legal directory>/legal/gnu_free_doc_1.2.pdf. Portions of this software may be subject to the Boost License version 1.0. You can view a copy of the Boost License v1.0 at: <path to legal directory>/legal/boost_1.0.pdf. To obtain a copy of the GNU GCC version 4.5.0 source code, send a request to request_sourcecode@mentor.com. This offer shall only be available for three years from the date Mentor Graphics Corporation first distributed GNU GCC version 4.5.0 and valid for as long as Mentor Graphics offers customer support for this Mentor Graphics product. GNU GCC version 4.5.0 may be subject to the following copyrights:

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **ORDERS, FEES AND PAYMENT.**

   1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.

   1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

   1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **BETA CODE.**

3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.

3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of this Agreement.

4. **RESTRICTIONS ON USE.**

4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer's product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.

4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

4.3. Customer agrees that it will not subject any Product to any open source software ("OSS") license that conflicts with this Agreement or that does not otherwise apply to such Product.

4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

4.5. The provisions of this Section 4 shall survive the termination of this Agreement.

5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at http://supportnet.mentor.com/supportterms.

6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations ("Third Party Terms") may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement

(including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of this Agreement.

7. **LIMITED WARRANTY.**

7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **LIMITATION OF LIABILITY.** TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **THIRD PARTY CLAIMS.**

9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.

9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.

9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

10. **INFRINGEMENT.**

10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

11. **TERMINATION AND EFFECT OF TERMINATION.**

11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.

13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.

14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.

16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 170330, Part No. 270941