Chapter 1. Avalon Verification IP Suite User Guide

Chapter 2. Avalon Memory-Mapped Master BFM

Functional Description ........................................................................................................ 2–1
  Timing ................................................................................................................................. 2–2
  Block Diagram ................................................................................................................... 2–5
Parameters ............................................................................................................................. 2–6
Application Program Interface (API) ................................................................................. 2–8
  all_transactions_complete() .............................................................................................. 2–8
  get_command_issued_queue_size() ..................................................................................... 2–8
  get_command_pending_queue_size() .................................................................................. 2–9
  get_response_address() .................................................................................................... 2–9
  get_response_byte_enable() .............................................................................................. 2–9
  get_response_burst_size() .................................................................................................. 2–9
  get_response_data() .......................................................................................................... 2–9
  get_response_latency() ...................................................................................................... 2–9
  get_response_request() ..................................................................................................... 2–10
  get_response_queue_size() ............................................................................................... 2–10
  get_response_wait_time() ................................................................................................. 2–10
  get_version() .................................................................................................................... 2–10
  init() ................................................................................................................................. 2–11
  pop_response() .................................................................................................................. 2–11
  push_command() ................................................................................................................ 2–11
  set_command_address() .................................................................................................... 2–11
  set_command_byte_enable() .............................................................................................. 2–11
  set_command_burst_count() .............................................................................................. 2–12
  set_command_burst_size() ................................................................................................. 2–12
  set_command_data() .......................................................................................................... 2–12
  set_command_idle() .......................................................................................................... 2–12
  set_command_init_latency() ............................................................................................. 2–13
  set_command_request() .................................................................................................... 2–13
  signal_all_transactions_complete() .................................................................................. 2–13
  signal_command_issued() ................................................................................................. 2–13
  signal_response_complete() ............................................................................................ 2–13

Chapter 3. Avalon Memory-Mapped Master BFM with Avalon-ST API Wrapper

Chapter 4. The Avalon Memory-Mapped Slave BFM

Functional Description ........................................................................................................ 4–2
  Timing ................................................................................................................................. 4–2
  Block Diagram ................................................................................................................... 4–4
Parameters ............................................................................................................................. 4–5
Application Program Interface (API) ........................................ 4–7
get_command_address() ........................................ 4–7
get_command_burst_count() ........................................ 4–7
get_command_burst_cycle() ........................................ 4–8
get_command_byte_enable() ........................................ 4–8
get_command_data() ........................................ 4–8
get_command_queue_size() ........................................ 4–8
get_command_request() ........................................ 4–8
get_response_queue_size() ........................................ 4–9
get_version() ........................................ 4–9
init() ........................................ 4–9
pop_command() ........................................ 4–9
push_response() ........................................ 4–9
set_command_transaction_mode() ........................................ 4–10
set_response_timeout() ........................................ 4–11
set_response_data() ........................................ 4–10
set_response_latency() ........................................ 4–11
set_response_timeout() ........................................ 4–11
signal_command_received ........................................ 4–11
signal_error_exceed_max_pending_reads ........................................ 4–11
signal_fatal_error ........................................ 4–11
enable_a_exclusive_read_write() ........................................ 6–6
enable_a_no_readdatavalid_during_reset() ........................................ 6–6
enable_a_less_than_burstcount_max_size() ........................................ 6–6
enable_a_half_cycle_reset_legal() ........................................ 6–6
enable_a_read_response_timeout() ........................................ 6–7
enable_a_waitrequest_timeout() ........................................ 6–7
enable_a_waitrequest_during_reset() ........................................ 6–7
enable_a_no_write_during_reset() ........................................ 6–7
enable_a_no_read_during_reset() ........................................ 6–7
enable_a_writeburst_timeout() ........................................ 6–8
enable_c_b2b_read_read() ........................................ 6–8
enable_c_b2b_read_write() ........................................ 6–8
enable_c_b2b_write_read() ........................................ 6–8
enable_c_b2b_write_write() ........................................ 6–8

Chapter 5. Avalon Memory-Mapped Slave BFM with Avalon-ST API Wrapper

Chapter 6. Avalon-MM Monitor

Parameters ........................................ 6–2
Application Program Interface (API) ........................................ 6–3
enable_a_beginbursttransfer_exist() ........................................ 6–4
enable_a_beginbursttransfer_legal() ........................................ 6–4
enable_a_beginbursttransfer_single_cycle() ........................................ 6–4
enable_a_begintransfer_exist() ........................................ 6–4
enable_a_begintransfer_legal() ........................................ 6–4
enable_a_begintransfer_single_cycle() ........................................ 6–5
enable_a_burst_legal() ........................................ 6–5
enable_a_byteenable_legal() ........................................ 6–5
enable_a_constant_during_burst() ........................................ 6–5
enable_a_constant_during_waitrequest() ........................................ 6–5
enable_a_exclusive_read_write() ........................................ 6–6
enable_a_half_cycle_reset_legal() ........................................ 6–6
enable_a_less_than_burstcount_max_size() ........................................ 6–6
enable_a_less_than_maximumpendingreadtransactions() ........................................ 6–6
enable_a_no_readdatavalid_during_reset() ........................................ 6–6
enable_a_read_response_timeout() ........................................ 6–7
enable_a_no_read_during_reset() ........................................ 6–7
enable_a_no_write_during_reset() ........................................ 6–7
enable_a_waitrequest_during_reset() ........................................ 6–7
enable_a_waitrequest_timeout() ........................................ 6–7
enable_a_writeburst_timeout() ........................................ 6–8
enable_c_b2b_read_read() ........................................ 6–8
enable_c_b2b_read_write() ........................................ 6–8
enable_c_b2b_write_read() ........................................ 6–8
enable_c_b2b_write_write() ........................................ 6–8
Chapter 7. Avalon Memory-Mapped BFM Tutorial
Overview of the Test Setup—Avalon-MM Slave DUT ........................................ 7–1
Setting Up the Test ........................................................................... 7–2
Running the Simulation ................................................................. 7–4
Observing the Results ................................................................. 7–5
Overview of the Test Setup—Avalon-MM Master DUT .................. 7–6
Setting Up the Test ........................................................................ 7–7
Running the Simulation ................................................................. 7–9
Observing the Results ................................................................. 7–9

Chapter 8. Avalon Streaming Source BFM
Functional Description .................................................................. 8–1
Timing ......................................................................................... 8–2
Block Diagram ........................................................................... 8–2
Parameters .................................................................................. 8–3
Application Program Interface (API) ............................................. 8–4
get_response_latency() ................................................................. 8–5
get_response_queue_size() ............................................................ 8–5
get_src_ready() ........................................................................ 8–5
get_src_transaction_complete() ................................................ 8–5
get_transaction_queue_size() ....................................................... 8–5
get_version() ........................................................................... 8–5
init() ......................................................................................... 8–6
pop_response() ........................................................................ 8–6
push_transaction() ..................................................................... 8–6
set_response_timeout() ............................................................... 8–6
set_transaction_channel() .......................................................... 8–6
set_transaction_data() ............................................................... 8–6
set_transaction_idles() .............................................................. 8–7
set_transaction_eop() ............................................................... 8–7
set_transaction_empty() ............................................................ 8–7
set_transaction_error() ............................................................. 8–7
set_transaction_sop() ............................................................... 8–7
signal_fatals_error ................................................................... 8–7
signal_response_done ................................................................. 8–8
signal_src_driving_transaction .................................................. 8–8
signal_src_not_ready ................................................................. 8–8
signal_src_ready .................................................................... 8–8
signal_src_transaction_complete ............................................. 8–8
Chapter 9. Avalon Streaming Source BFM with Avalon-ST API Wrapper

Chapter 10. Avalon Streaming Sink BFM
  Functional Description .......................................................... 10–1
  Timing ............................................................................... 10–2
  Block Diagram ..................................................................... 10–2
  Parameters ......................................................................... 10–3
  Application Program Interface (API) ...................................... 10–4
  get_transaction_channel() .................................................... 10–4
  get_transaction_data() ........................................................... 10–5
  get_transaction_idles() .......................................................... 10–5
  get_transaction_eop() .............................................................. 10–5
  get_transaction_empty() ........................................................ 10–5
  get_transaction_error() ............................................................. 10–5
  get_transaction_queue_size() .................................................. 10–5
  get_transaction_sop() ............................................................... 10–6
  get_version() ..................................................................... 10–6
  init() ............................................................................... 10–6
  pop_transaction() ................................................................. 10–6
  set_ready() ...................................................................... 10–6
  signal_fatal_error ................................................................. 10–6
  signal_sink_ready ............................................................... 10–7
  signal_sink_ready_deassert .................................................... 10–7
  signal_transaction_received .................................................. 10–7
  signal_transaction_received .................................................. 10–7

Chapter 11. Avalon Streaming Sink BFM with Avalon-ST API Wrapper

Chapter 12. Avalon-ST Monitor
  Parameters ......................................................................... 12–2
  Application Program Interface (API) ...................................... 12–2
  enable_a_empty_legal() ......................................................... 12–3
  enable_a_less_than_max_channel() ....................................... 12–3
  enable_a_no_data_outside_pack() ......................................... 12–3
  enable_a_non_missing_endofpacket() ..................................... 12–4
  enable_a_non_missing_startofpacket() .................................... 12–4
  enable_a_valid_legal() ........................................................... 12–4
  enable_b2b_data_different_channel() .................................... 12–4
  enable_b2b_data_same_channel() ........................................... 12–4
  enable_b2b_packet_different_channel() .................................. 12–5
  enable_b2b_packet_same_channel() ....................................... 12–5
  enable_c_channel_change_in_packet() .................................... 12–5
  enable_c_empty() ................................................................ 12–5
  enable_c_error() ................................................................ 12–5
  enable_c_non_valid_ready() ................................................... 12–6
  enable_c_non_valid_non_ready() ............................................. 12–6
  enable_c_packet() ................................................................. 12–6
  enable_c_packet_no_idles_no_back_pressure() ....................... 12–6
  enable_c_packet_with_back_pressure() ................................... 12–6
  enable_c_packet_with_idles() ................................................ 12–7
  enable_c_transfer() ............................................................... 12–7
  enable_c_valid_non_ready() .................................................... 12–7
Additional Information

Revision History .............................................................. Info–1
How to Contact Altera ......................................................... Info–1
Typographic Conventions ................................................... Info–1
The Avalon® Verification IP Suite provides bus functional models (BFMs) to simulate the behavior of Avalon Memory-Mapped (Avalon-MM) master and slave interfaces and Avalon Streaming (Avalon-ST) source and sink interfaces. It also provides monitors to verify both Avalon protocols. This suite facilitates the verification of IP that includes Avalon-MM and Avalon-ST interfaces. Using the Altera-provided BFMs and monitors has the following advantages:

- It accelerates the verification process by providing key components of the verification testbench.
- It provides Avalon BFM components that implement the standard Avalon-MM and Avalon-ST protocols, serving as a reference for those protocols.
- For SystemVerilog users, it provides a platform that you can use to implement constraint-driven randomized tests, including traffic scenario drivers, scoreboard and coverage facilities, and assertion checkers.

Figure 1–1 shows the top-level blocks in a typical testbench to verify components with Avalon-MM and Avalon-ST interfaces.
Figure 1–1 shows the top-level blocks in a testbench to verify components with Avalon-MM and Avalon-ST interfaces. As Figure 1–1 illustrates, it is possible to write a testbench using a traditional Verilog HDL implementation or using SystemVerilog with VMM. For purposes of illustration, Figure 1–1 shows an Avalon-MM DUT that includes both Avalon-MM master and slave interfaces, and an Avalon-ST DUT that includes both source and sink interfaces, although typical components might include a single Avalon interface.

When verifying a component with Avalon-MM or Avalon-ST interfaces, a monitor is interposed between the master or source BFM and the slave or sink interface of the DUT. A second monitor can be interposed between the slave or sink BFM and the master or source interface of the DUT. The monitors do not have to be placed between a BFM component and another component. They can be inserted anywhere in the system to provide protocol assertion checking and functional coverage reporting.

The test program drives the stimulus to the DUTs and determines whether the DUTs’ behavior is correct, by analyzing the responses. The BFM components translate the test program stimuli, creating the signalling for the Avalon-MM and Avalon-ST protocols. The monitors verify Avalon protocol compliance and provide test coverage reports.

The APIs for the Avalon Verification IP Suite components include the methods to construct all of the Avalon-MM and Avalon-ST transactions.

The Avalon Verification IP Suite BFMs are implemented in SystemVerilog. The BFM components use primarily Verilog HDL with a few basic SystemVerilog constructs which are supported by ModelSim® AE. The monitor components use the SystemVerilog Assertion (SVA) language and are supported only by simulators that support SVA, including: Modelsim® SE, Synopsis VCS, and Mentor Questa.

The Avalon Verification IP Suite also includes wrapper components so that the BFMs, which are implemented in SystemVerilog, can be used in VHDL verification environments. You can include the BFMs with wrappers in simulators that support mixed language simulation.

This Avalon Verification IP Suite User Guide provides a reference document for the each of the BFMs and Avalon Monitors. It includes the following chapters:

Chapter 2, Avalon Memory-Mapped Master BFM
Chapter 3, Avalon Memory-Mapped Master BFM with Avalon-ST API Wrapper
Chapter 4, The Avalon Memory-Mapped Slave BFM
Chapter 5, Avalon Memory-Mapped Slave BFM with Avalon-ST API Wrapper
Chapter 6, Avalon-MM Monitor
Chapter 8, Avalon Streaming Source BFM
Chapter 9, Avalon Streaming Source BFM with Avalon-ST API Wrapper
Chapter 10, Avalon Streaming Sink BFM
Chapter 11, Avalon Streaming Sink BFM with Avalon-ST API Wrapper
Chapter 12, Avalon-ST Monitor
The Avalon® Memory-Mapped (Avalon-MM) Master Bus Functional Model (BFM) implements the Avalon-MM interface protocol. This BFM component is controlled via an application programming interface (API). The API consists of procedures that construct transactions which correspond to the Avalon-MM protocol, including: read, write, burst read, and burst write. Figure 2–1 shows the top-level modules for a typical testbench that uses the Avalon-MM BFM to verify an Avalon-MM slave component. In addition to the Altera-provided Avalon-MM Master BFM component, the typical testbench includes a test program and the DUT which includes an Avalon-MM slave interface. Using the Altera-provided Avalon-MM BFM has two significant advantages:

- It accelerates development by supplying one of the modules for the verification testbench.
- It provides an independent check on the DUT implementation of the Avalon-MM slave protocol. Because the testbench includes an Avalon-MM master designed by Altera and an Avalon-MM slave designed by the user, it should highlight any misinterpretation of the protocol implemented by the DUT which might be missed in a testbench designed by a single engineer.

The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitors components verify protocol compliance.

For more information about the Avalon-MM specification refer to the Avalon Interface Specifications.
Functional Description

This section provides a functional description of the Avalon-MM Master BFM. It includes the following topics:

“Timing” on page 2–2
“Block Diagram” on page 2–6

Timing

The timing diagram shown in Figure 2–2 illustrates the sequence of events for an Avalon-MM Master BFM driving interleaved writes and reads when the readdatavalid signal is present and serves as a reference for the following discussion of APIs and events.
Table 2–1 explains the annotations used in Figure 2–2.

Table 2–1. Key to Annotations in Figure 2–2

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_{init}</td>
<td>The initial command latency which is 2 cycles for transactions 1 and 2. This time is set by the API command set_command_init_latency.</td>
</tr>
<tr>
<td>T_{wt,1}</td>
<td>The response wait time, which is 3 cycles. This time is determined by the number of cycles that the waitrequest signal is asserted by the slave. The program gets this value using the get_response_wait_time command.</td>
</tr>
<tr>
<td>T_{wr}</td>
<td>waitrequest is always sampled #1 after the falling edge of clk.</td>
</tr>
</tbody>
</table>
The timing diagram shown in Figure 2–3 illustrates the sequence of events for an Avalon-MM Master BFM driving a write followed by a read when the readdatavalid signal is not present.

### Table 2–1. Key to Annotations in Figure 2–2

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{idle}}$</td>
<td>The idle time after each transaction. This time is set by the command <code>set_command_idle</code>.</td>
</tr>
</tbody>
</table>
| $T_{\text{rl}1}$ | The response latency for the first read which is 3 cycles. This is the time between when the read command is accepted, and the read response is provided by the slave. The program gets this time using the `get_response_latency` command.  
  Note if the Avalon-MM slave component has defined a fixed read latency by defining the `readLatency` interface property, the `readdatavalid` signal is not used. For more information refer to the `Avalon Interface Specifications`. |
| $T_{\text{rl}2}$ | The response latency for the first read which is 3 cycles. The program gets this time using the `get_response_latency` command.                   |
| $S_{CI1}$–$S_{CI4}$ | Signals when write or read commands are presented on the interface. The event name is `signal_command_issued`.                              |
| $S_{RC2}$–$S_{RC4}$ | Signals read responses. The event name is `signal_response_complete`.                                                                         |
| $S_{\text{all}}$ | Signals the end of the test. The event name is `signal_all_transactions_complete`                                                          |
Table 2–2 explains the annotations used in Figure 2–3.

### Table 2–2. Key to Annotations in Figure 2–3

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{init}}$</td>
<td>The initial command latency which is 2 cycles for transactions 1 and 2. This time is set by the API command <code>set_command_init_latency</code>.</td>
</tr>
<tr>
<td>$T_{\text{wait}_1}$</td>
<td>The response wait time, which is 3 cycles. This time is determined by the number of cycles that the <code>waitrequest</code> signal is asserted by the slave. The program gets this value using the <code>get_response_wait_time</code> command.</td>
</tr>
<tr>
<td>$T_{\text{wr}}$</td>
<td><code>waitrequest</code> is always sampled #1 after the falling edge of <code>clk</code>.</td>
</tr>
<tr>
<td>$T_{\text{idle}}$</td>
<td>The idle time after a transaction. This time is set by the command <code>set_command_idle</code>.</td>
</tr>
<tr>
<td>$S_{\text{cl}<em>1}$–$S</em>{\text{cl}_2}$</td>
<td>Signals when write and read commands are presented on the interface. The event name is <code>signal_command_issued</code>.</td>
</tr>
<tr>
<td>$S_{\text{rc}_1}$</td>
<td>Signals the first read response. The event name is <code>signal_response_complete</code>.</td>
</tr>
<tr>
<td>$S_{\text{allc}}$</td>
<td>Signals the end of the test. The event name is <code>signal_all_transactions_complete</code>.</td>
</tr>
</tbody>
</table>
Figure 2–4 provides a block diagram of the Avalon-MM Master BFM. As this figure illustrates, the BFM includes the following six major blocks.

- **Avalon-MM Master API**—Provides methods to create Avalon-MM transactions and query the state of all queues.

- **Command Descriptor**—Accumulates the fields of an Avalon-MM command transaction using the `set_command` API calls and pushes completed commands onto the pending command queue.

- **Avalon-MM Interface Driver**—Issues transfers to the system interconnect fabric and holds each transfer until `waitrequest` is deasserted. For burst transfers, there is a separate transfer for each word of the burst. The system interconnect fabric can assert `waitrequest` for each word of the burst, as necessary.

- **Timestamp Counter**—Records a timestamp with commands for use in timing calculations. The driver and monitor both use the timestamp counter for timing calculations.

- **Avalon-MM Interface Monitor**—Monitors the system interconnect fabric and records responses for read transfers in the response queue.

- **Response Descriptor**—Collects information about completed transactions using the `get_response` API calls. The testbench uses this information for further analysis.
Parameters

The Avalon-MM BFM supports the full range of signals defined for the Avalon-MM master interface. You can customize the Avalon-MM master interface using the parameters described in Table 2–3.

For detailed information about the Avalon-MM interface, including timing diagrams, refer to the *Avalon Interface Specifications*. 
### Table 2–3. Parameters for the Avalon-MM Master BFM  (Part 1 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Widths</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address width</td>
<td>32</td>
<td>Address width in bits.</td>
<td></td>
</tr>
<tr>
<td>Symbol width</td>
<td>8</td>
<td>Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.</td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of symbols</td>
<td>4</td>
<td>Numbers of symbols per word.</td>
<td></td>
</tr>
<tr>
<td>Burstcount width</td>
<td>3</td>
<td>The width of the burst count in bits.</td>
<td></td>
</tr>
<tr>
<td><strong>Port Enables</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use the read signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a read pin.</td>
</tr>
<tr>
<td>Use the write signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a write pin.</td>
</tr>
<tr>
<td>Use the address signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes address pins.</td>
</tr>
<tr>
<td>Use the byteenable signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes byte enable pins.</td>
</tr>
<tr>
<td>Use the burstcount signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes burstcount pins.</td>
</tr>
<tr>
<td>Use the readdata signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a readdata pin.</td>
</tr>
<tr>
<td>Use the readdatavalid signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a readdatavalid pin.</td>
</tr>
<tr>
<td>Use the writedata signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a writedata pin.</td>
</tr>
<tr>
<td>Use the begintransfer signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes begintransfer pins.</td>
</tr>
<tr>
<td>Use the beginbursttransfer signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes a beginbursttransfer pins.</td>
</tr>
<tr>
<td>Use the waitrequest signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a waitrequest pin.</td>
</tr>
<tr>
<td><strong>Port Polarity</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assert reset high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, reset is asserted high.</td>
</tr>
<tr>
<td>Assert waitrequest high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, waitrequest is asserted high.</td>
</tr>
<tr>
<td>Assert read high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, read is asserted high.</td>
</tr>
<tr>
<td>Assert write high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, write is asserted high.</td>
</tr>
<tr>
<td>Assert byteenable high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, byteenable is asserted high.</td>
</tr>
<tr>
<td>Assert readdatavalid high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, readdatavalid is asserted high.</td>
</tr>
<tr>
<td><strong>Burst Attributes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linewrap burst</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the address for bursts wraps instead of an incrementing. With a wrapping burst, when the address reaches a burst boundary, it wraps back to the previous burst boundary such that only the low order bits need to be used for addressing.</td>
</tr>
<tr>
<td>Burst on burst boundaries only</td>
<td>On</td>
<td>On/Off</td>
<td>When On, memory bursts are aligned to the address size.</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum pending reads</td>
<td>1</td>
<td>The maximum number of pending reads which can be queued by the slave.</td>
<td></td>
</tr>
<tr>
<td>Fixed read latency (cycles)</td>
<td>1</td>
<td>Sets the read latency for fixed-latency slaves. Not used on interfaces that include the readdatavalid signal.</td>
<td></td>
</tr>
</tbody>
</table>
Application Program Interface (API)

The API provides methods for a testbench which instantiates, controls and queries state in this BFM component. Test programs must only use these public access methods and events to communicate with this BFM component. While it is possible to use the methods that are not included in the public API published below, Altera does not guarantee continued support or backwards compatibility of the private methods.

The API includes methods that fall into the three categories:
- Methods to initialize the BFM and determine its state
- Methods to create Avalon-MM reads and writes
- Methods to determine the response of the DUT

Chapter 7, Avalon Memory-Mapped BFM Tutorial illustrates how you can use these commands to test an Avalon-MM slave device. The following section presents these methods in alphabetical order.

### all_transactions_complete()

**Prototype:**

```
int all_transactions_complete().
```

**Arguments:**

None.

**Returns:**

Int.

**Description:**

Queries the BFM component to determine whether all issued commands have been completed. A return value of 1 means that there are no more transactions in the transaction queue or in progress.

### get_command_issued_queue_size()

**Prototype:**

```
int get_command_issued_queue_size().
```

**Arguments:**

None.

**Returns:**

int.

**Description:**

Queries the issued command queue to determine the number of commands that have been driven to the system interconnect fabric, but not completed.

---

**Table 2–3. Parameters for the Avalon-MM Master BFM (Part 2 of 2)**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Streaming Interface (Note 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width of API interface data signal</td>
<td>64</td>
<td>64</td>
<td>The width of the data signal.</td>
</tr>
<tr>
<td>Width of API return interface data</td>
<td>64</td>
<td>64</td>
<td>The width of the return interface data signal.</td>
</tr>
</tbody>
</table>

**Note to Table 2–3:**

(1) This interface is only required for the Avalon-MM Master BFM with Avalon-ST API Wrapper which is used in mixed language simulations.
get_command_pending_queue_size()

Prototype:   
int get_command_pending_queue_size().
Arguments:   None.
Returns:     int.
Description: Queries the command queue to determine number of pending commands waiting to be driven out as Avalon requests.

get_response_address()

Prototype:   
bit [AV_ADDRESS_W-1:0] get_response_address().
Arguments:   None.
Returns:     bit.
Description: Returns the transaction address in the response descriptor that has been popped from the response queue.

get_response_byte_enable()

Prototype:   
bit [AV_NUMSYMBOLS-1:0] get_response_byte_enable(int index).
Arguments:   index.
Returns:     bit.
Description: Returns the value of the byte enables in the response descriptor that has been popped from the response queue. Each cycle of a burst response is addressed individually by the specified index.

get_response_burst_size()

Prototype:   
bit [AV_BURSTCOUNT_W-1:0] get_response_burst_size ().
Arguments:   None.
Returns:     bit.
Description: Returns the size of the response transaction burst count in the response descriptor that has been popped from the response queue.

get_response_data()

Prototype:   
bit [AV_DATA_W-1:0] get_response_data(int index).
Arguments:   index.
Returns:     bit.
Description: Returns the transaction read data in the response descriptor that has been popped from the response queue. Each cycle in a burst response is addressed individually by the specified index. In the case of read responses, the data is the data captured on the avm_readdata interface pin. In the case of write responses, the data on the driven avm_writedata pin is captured and reflected here.
get_response_latency()

Prototype: int get_response_latency(int index).
Arguments: index.
Returns: bit.
Description: Returns the transaction read latency in the response descriptor that has been popped from the response queue. Each cycle in a burst read has its own latency entry. For write transaction responses the returned value will always be 0.

get_response_request()

Prototype: Request_t get_response_request().
Arguments: None.
Returns: Request_t.
Description: Returns the transaction command type in the response descriptor that has been popped from the response queue.

get_response_queue_size()

Prototype: int get_response_queue_size().
Arguments: None.
Returns: automatic int.
Description: Queries the response queue to determine number of response descriptors currently stored in the BFM. This is the number of responses the test program can immediately pop off the response queue for further processing.

get_response_wait_time()

Prototype: int get_response_wait_time(int index).
Arguments: index.
Returns: automatic int.
Description: Returns the wait latency for transaction in the response descriptor that has been popped from the response queue. Each cycle in a burst has its own wait latency entry.

get_version()

Prototype: string get_version().
Arguments: None.
Returns: String.
Description: Return BFM version as a string of three integers separated by periods. For example, version 9.1 sp1 is encoded as ’9.1.1’.
init()

Prototype: init.
Arguments: None.
Returns: void.
Description: Initializes the Avalon-MM Master interface.

pop_response()

Prototype: void pop_response();
Arguments: None.
Returns: void.
Description: Pop the oldest response descriptor from the response queue, such that transaction information is available using the get_response_* commands.

push_command()

Prototype: void push_command();
Arguments: None.
Returns: void.
Description: Pushes the fully populated transaction descriptor onto the pending transaction command queue.

set_command_address()

Prototype: void set_command_address(bit[AV_ADDRESS_W-1:0] addr);
Arguments: addr.
Returns: void.
Description: Sets the transaction address in the command descriptor.

set_command_byte_enable()

Prototype: void set_command_byte_enable(bit[AV_NUMSYMBOLS-1:0] byte_enable, int index);
Arguments: byte_enable.
index.
Returns: void.
Description: Sets the transaction byte enable field for the cycle of the burst command descriptor indicated by index. This field applies to both read and write operations.
**set_command_burst_count()**

Prototype: 
```c
```

Arguments: 
- `burst_count`.

Returns: 
- `void`.

Description: 
Sets the value driven on the Avalon interface `burstcount` pin. Generates a warning message if the specified `burst_count` is out of range. Not available if the `USE_BURSTCOUNT` parameter is false.

**set_command_burst_size()**

Prototype: 
```c
void set_command_burst_size (bit[AV_BURSTCOUNT_W-1:0] burst_size).
```

Arguments: 
- `burst_size`.

Returns: 
- `void`.

Description: 
Sets the transaction burst count in the command descriptor to determine the number of words driven on the write burst command. The value may be different from the value specified in `set_command_burst_count` to generate illegal traffic for testing. Generates a warning if the value is different.

**set_command_data()**

Prototype: 
```c
void set_command_data(bit[AV_DATA_W-1:0] data, int index).
```

Arguments: 
- `data`.
- `index`.

Returns: 
- `void`.

Description: 
Sets the transaction write data in the command descriptor. For burst transactions, the command descriptor holds an array of data, with each element individually set by this method.

**set_command_idle()**

Prototype: 
```c
void set_command_idle(bit idle, int index).
```

Arguments: 
- `idle`.
- `index`.

Returns: 
- `void`.

Description: 
Set idle cycles at the end of each transaction cycle. In the case of read commands, idle cycles are inserted at the end of the command cycle. In the case of burst write commands, idle cycles are inserted at the end of each write data cycle within the burst.
**set_command_init_latency()**

Prototype: void set_command_init_latency(int cycles).
Arguments: cycles.
Returns: void.
Description: Set the number of cycles to postpone the start of a command.

**set_command_request()**

Prototype: void set_command_request(Request_t).
Arguments: Request_t.
Returns: void.
Description: Sets the transaction type to read or write in the command descriptor. The enumeration type defines \texttt{REQ\_READ} = 0 and \texttt{REQ\_WRITE} = 1.

**signal_all_transactions_complete()**

Prototype: signal_all_transactions_complete.
Arguments: None.
Returns: void.
Description: This event signals that all queued transactions have completed.

**signal_command_issued()**

Prototype: signal_command_issued.
Arguments: None.
Returns: void.
Description: This event signals that the currently pending command has been driven to the interface.

**signal_response_complete()**

Prototype: signal_response_complete.
Arguments: None.
Returns: void.
Description: This event signals that the read response has been received and pushed into the response queue.
The Avalon-MM Master BFM with Avalon-ST API Wrapper provides VHDL support for the Avalon-MM Master BFM. You can use the Avalon-MM Master BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation. This component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-MM Master BFM’s method calls and returns into signals that are carried on the call and return interface ports. To call a method, the method identifier is pushed into the BFM wrapper component via the channel field; the data is the arguments for the method. Once the method is complete, the data field transports the arguments for the method call. The response is returned on the response Avalon ST interface, and that Avalon ST data signal carries the return value. The wrapper is necessary because VHDL can only access ports and does not support the method calls across hierarchical boundaries used in the Avalon-MM Master BFM field. Figure 3–1 provides a high-level view the VHDL testbench communicating with the BFM.

**Figure 3–1.** Avalon-MM Master BFM with Avalon-ST Wrapper

In Figure 3–1, the API call interface and Avalon-ST call and return interface operate in separate clock domains with \( \text{av clk} \) synchronizing the FPGA logic and \( \text{api clk} \) synchronizing the Avalon-ST translation interface. Typically, the Avalon-ST interface, which is not part of the actual hardware design, operates at much higher frequencies than the Avalon-MM Slave BFM interface, enabling 1000 API calls and returns to be issued to the BFM per Avalon clock cycle.
For every function call in the BFM, there is a channel identifier that stores the fixed mapping between channel number and the function.

`<install_dir>/ip/altera/sopc_builder_ip/verification/lib/altera_avalon_components_pkg.vhd` defines the following function calls:

- MM_MSTR_INIT
- MM_MSTR_SET_RESP_TIMEOUT
- MM_MSTR_SET_CMD_TIMEOUT
- MM_MSTR_ALL_TRANS_COMPLETE
- MM_MSTR_GET_CMD_ISSUE_QUEUE_SIZE
- MM_MSTR_GET_CMD_PEND_QUEUE_SIZE
- MM_MSTR_GET_RESP_QUEUE_SIZE
- MM_MSTR_PUSH_CMD
- MM_MSTR_POP_RESP
- MM_MSTR_SET_CMD_DATA
- MM_MSTR_SET_CMD_ADDRESS
- MM_MSTR_SET_CMD_BYTE_ENABLE
- MM_MSTR_SET_CMD_BURST_COUNT
- MM_MSTR_SET_CMD_IDLE
- MM_MSTR_SET_CMD_REQUEST
- MM_MSTR_SET_CMD_RESERVED_1
- MM_MSTR_GET RESP REQUEST
- MM_MSTR_GET RESP DATA
- MM_MSTR_GET RESP ADDRESS
- MM_MSTR_GET RESP BYTE ENABLE
- MM_MSTR_GET RESP BURST_SIZE
- MM_MSTR_GET RESP LATENCY
- MM_MSTR_GET RESP WAIT TIME
- MM_MSTR_SET_CMD_INIT_LATENCY
- MM_MSTR_SET_CMD_BURST_SIZE

With the exception of the API wrapper which provides an alternative way to access the Avalon-MM Master BFM API to support VHDL testbenches, this component is identical to the Avalon-MM Master BFM. For more information about this component refer the Chapter 2, Avalon Memory-Mapped Master BFM.
The Avalon-MM Slave BFM implements the slave side of the Avalon-MM interface protocol. This is a standard memory-mapped protocol including reads and writes typical of simple peripherals and the reads, writes, burst reads, and burst writes for typical memory devices. This BFM also includes a procedural interface to monitor incoming commands, pass these to the test program, accept response transactions from the test program, and drive responses. Figure 4–1 shows the top-level modules for a testbench that uses the Avalon-MM Slave BFM to verify an Avalon-MM Master device. In addition to the Altera-provided Avalon-MM Slave BFM, the example testbench shown in Figure 4–1 includes a test program and the device under test (DUT). The test program, written in HDL, programs the Avalon-MM master to issue Avalon-MM transactions, programs the Avalon-MM Slave BFM to respond, and analyzes the results. Using the Altera-provided Avalon-MM Slave BFM has two significant advantages:

- It accelerates development by supplying one of the modules for the verification testbench.
- It provides an independent check on the DUT implementation of the Avalon-MM master protocol. Because the testbench includes an Avalon-MM slave designed by Altera and an Avalon-MM master designed by the user, it should highlight any misinterpretation of the protocol implemented by the DUT which might be missed in a testbench designed by a single engineer.

The BFMs allow illegal response transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitors components verify protocol compliance.

**Figure 4–1. Top-Level Module to Verify an Avalon-MM Master**
For more information about the Avalon-MM specification refer to the *Avalon Interface Specifications*.

**Functional Description**

This section provides a functional description of the Avalon-MM Master BFM. It includes the following topics:

“Timing” on page 4–2

“Block Diagram” on page 4–5

**Timing**

The timing diagram shown in Figure 4–2 illustrates the sequence of events for an Avalon-MM Slave BFM responding to interleaved writes and reads when the `readdatavalid` signal is present.
Table 4–1 explains the annotations used in Figure 4–2.

**Table 4–1. Key to Annotations in Figure 4–2** (Part 1 of 2)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{wt_1}$</td>
<td>The response wait time which is 3 cycles. The slave sets this value using the <code>set_interface_wait_time</code> command.</td>
</tr>
<tr>
<td>$T_\text{wr}$</td>
<td><code>waitrequest</code> is sampled #1 after the falling edge of <code>clk</code>.</td>
</tr>
<tr>
<td>$T_{wl_2}$</td>
<td>The response latency for the first read which is 3 cycles. The slave sets this value using the <code>set_interface_wait_time</code> command.</td>
</tr>
</tbody>
</table>
The timing diagram shown in Figure 4–3 illustrates the sequence of events for an Avalon-MM Slave BFM receiving a write followed by a read when the readdatavalid signal is not present.

**Figure 4–3.** Avalon-MM Slave Receiving Write and Read Commands with No readdatavalid Signal

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_{cr,1})–(S_{cr,2})</td>
<td>Signals when read commands were received. The event name is <code>signal_command_received</code>.</td>
</tr>
<tr>
<td>(T_{rl,1}), (T_{rl,2})</td>
<td>The response latency for the reads which is 3 cycles. The slave sets this time using the <code>set_response_latency</code> command.</td>
</tr>
</tbody>
</table>
Table 4–2 explains the annotations used in Figure 4–3.

**Table 4–2. Key to Annotations in Figure 4–3**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_I$</td>
<td>The initial command latency which is 2 cycles for transactions 1 and 2.</td>
</tr>
<tr>
<td>$T_{wt,1}$</td>
<td>The response wait time which is 3 cycles. The master gets this value using the <code>get_response_wait_time</code> command.</td>
</tr>
<tr>
<td>$T_{wr}$</td>
<td><code>waitrequest</code> is sampled #1 after the falling edge of <code>clk</code>.</td>
</tr>
<tr>
<td>$T_{rl,1}$</td>
<td>The response latency for the first read which is 2 cycles. The master gets this time using the <code>get_response_latency</code> command.</td>
</tr>
<tr>
<td>$S_{cl,1}, S_{cl,2}$</td>
<td>Signals write and read commands. The event name is <code>signal_command_issued</code>.</td>
</tr>
<tr>
<td>$S_{rc,1}$</td>
<td>Signals the first read response. The event name is <code>signal_response_complete</code>.</td>
</tr>
<tr>
<td>$S_{bc}$</td>
<td>Signals the end of the test. The event name is <code>signal_all_transactions_complete</code>.</td>
</tr>
</tbody>
</table>

**Block Diagram**

Figure 4–4 provides a block diagram of the Avalon-MM Slave BFM. As this figure illustrates, the BFM includes the following five major blocks.

- **Avalon-MM Slave API**—Provides methods to get commands and create responses to commands from the Avalon-MM master which is the DUT.

- **Command Descriptor**—Accumulates the fields of a command sent by the Avalon-MM master and sends completed commands to the Avalon-MM Slave BFM when requested.

- **Avalon-MM Interface Monitor**—Monitors activity coming from the Avalon-MM Master (DUT) and stores commands in the Client Command Queue.

- **Response Generator and Data Cache**—In `memory_mode` the Slave BFM models a single port RAM. A write operation stores the data in an associative array and generates no response. A read operation fetches data from the array and drives it on the response side of the Avalon interface. This mode simplifies loopback testing.

- **Avalon-MM Slave Interface Driver**—Drives responses to the system interconnect fabric. For burst transfers, there is a separate transfer for each word of the burst. The client testbench can instruct the Slave BFM to assert `waitrequest` for each word of the burst to test the functionality of the Avalon-MM master.
The Avalon-MM Slave BFM supports the full range of signals defined for the Avalon-MM slave interface. You can customize the Avalon-MM slave interface using the parameters described in Table 4–3.

For detailed information about the Avalon-MM interface, including timing diagrams, refer to the Avalon Interface Specifications.
### Table 4–3. Parameters for the Avalon-MM Slave BFM (Part 1 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Widths</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address width</td>
<td>32</td>
<td>Address width in bits.</td>
<td></td>
</tr>
<tr>
<td>Symbol width</td>
<td>8</td>
<td>Data symbol width in bits. Set AV_SYMBOL_W to 8 for byte-oriented interfaces.</td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of symbols</td>
<td>4</td>
<td>Numbers of symbols per word.</td>
<td></td>
</tr>
<tr>
<td>Burstcount width</td>
<td>3</td>
<td>The width of the burst count in bits.</td>
<td></td>
</tr>
<tr>
<td><strong>Port Enables</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use the read signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a read pin.</td>
</tr>
<tr>
<td>Use the write signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a write pin.</td>
</tr>
<tr>
<td>Use the address signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes address pins.</td>
</tr>
<tr>
<td>Use the byte enable signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes byte_enable pins.</td>
</tr>
<tr>
<td>Use the burstcount signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes burstcount pins.</td>
</tr>
<tr>
<td>Use the readdata signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a readdata pin.</td>
</tr>
<tr>
<td>Use the readdatavalid signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a readdatavalid pin.</td>
</tr>
<tr>
<td>Use the writedata signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a writedata pin.</td>
</tr>
<tr>
<td>Use the begintransfer signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes a begintransfer pins.</td>
</tr>
<tr>
<td>Use the beginbursttransfer signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes a beginbursttransfer pins.</td>
</tr>
<tr>
<td>Use the waitrequest signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a waitrequest pin.</td>
</tr>
<tr>
<td><strong>Port Polarity</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assert reset high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, reset is asserted high.</td>
</tr>
<tr>
<td>Assert waitrequest high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, waitrequest is asserted high.</td>
</tr>
<tr>
<td>Assert read high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, read is asserted high.</td>
</tr>
<tr>
<td>Assert write high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, write is asserted high.</td>
</tr>
<tr>
<td>Assert byteenable high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, byteenable is asserted high.</td>
</tr>
<tr>
<td>Assert readdatavalid high</td>
<td>On</td>
<td>On/Off</td>
<td>When On, readdatavalid is asserted high.</td>
</tr>
<tr>
<td><strong>Burst Attributes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linewrap burst</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the address for bursts wraps instead of incrementing. With a wrapping burst, when the address reaches a burst boundary, it wraps back to the previous burst boundary such that only the low order bits need to be used for addressing.</td>
</tr>
<tr>
<td>Burst on burst boundaries only</td>
<td>On</td>
<td>On/Off</td>
<td>When On, memory bursts are aligned to the address size.</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum pending reads</td>
<td>1</td>
<td></td>
<td>The maximum number of pending reads which can be queued up by the slave.</td>
</tr>
<tr>
<td>Fixed read latency (cycles)</td>
<td>0</td>
<td></td>
<td>Sets the read latency for fixed-latency slaves. Not used on interfaces that include the readdatavalid signal.</td>
</tr>
</tbody>
</table>
The API includes methods that fall into the three categories:

- Methods to initialize the BFM and determine its state
- Methods to get commands from the Avalon-MM Master (DUT)
- Methods to create responses to the Avalon-MM Master (DUT)

Chapter 7, Avalon Memory-Mapped BFM Tutorial illustrates how you can use these commands to test an Avalon-MM master device.

### get_command_address()

**Prototype:**

```c
bit [AV_ADDRESS_W-1:0] get_command_address();
```

**Arguments:** None.

**Returns:** `bit [AV_ADDRESS_W-1:0]`.

**Description:** Queries the received command descriptor for the transaction address.

### get_command_burst_count()

**Prototype:**

```c
[AV_BURSTCOUNT_W-1:0] get_command_burst_count();
```

**Arguments:** None.

**Returns:** `[AV_BURSTCOUNT_W-1:0]`.

**Description:** Queries the received command descriptor for the transaction burst count.

---

**Table 4–3. Parameters for the Avalon-MM Slave BFM (Part 2 of 2)**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed read wait time (cycles)</td>
<td>1</td>
<td>Fixed range</td>
<td>For slave interfaces that don’t use the <code>waitrequest</code> signal, the read wait time indicates the number of cycles before the slave responds to a read. The timing is as if the slave asserted <code>waitrequest</code> for this number of cycles.</td>
</tr>
<tr>
<td>Fixed write wait time (cycles)</td>
<td>0</td>
<td>Fixed range</td>
<td>For slave interfaces that don’t use the <code>waitrequest</code> signal, the write wait time indicates the number of cycles before the slave accepts a write.</td>
</tr>
</tbody>
</table>

API Streaming Interface *(Note 1)*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width of API interface data signal</td>
<td>64</td>
<td>Width of data signal.</td>
<td></td>
</tr>
<tr>
<td>Width of API return interface data signal</td>
<td>64</td>
<td>Width of return interface data signal.</td>
<td></td>
</tr>
</tbody>
</table>

*(Note 1)* This interface is only required for the Avalon-MM Slave BFM with Avalon-ST API Wrapper which is used in mixed language simulations.
get_command_burst_cycle()

Prototype: int get_command_burst_cycle().
Arguments: None.
Returns: Int.
Description: The slave BFM receives and processes write burst commands as a sequence of discrete commands. The number of commands corresponds to the burst count. A separate command descriptor is constructed for each write burst cycle, corresponding to a partially completed burst. This method returns a burst cycle field that tells the testbench which burst cycle was active when this descriptor was constructed. This facility enables the testbench to query partially completed write burst operations. In other words, the testbench can query the write data word on each burst cycle as it arrives and begin to process it immediately rather than waiting until the entire burst has been received, making it possible to perform pipelined write burst processing in the testbench.

get_command_byte_enable()

Prototype: bit [AV_NUMSYMBOLS-1:0] get_command_byte_enable (int index).
Arguments: index.
Returns: int.
Description: Queries the received command descriptor for the transaction byte enable. For burst commands with burst count greater than 1, the index selects the data cycle.

get_command_data()

Prototype: bit [AV_DATA_W-1:0] get_command_data(int index).
Arguments: index.
Returns: int.
Description: Queries the received command descriptor for the transaction write data. For burst commands with burst count greater than 1, the index selects the write data cycle.

get_command_queue_size()

Prototype: int get_command_queue_size().
Arguments: None.
Returns: int.
Description: Queries the command queue to determine number of pending commands.

get_command_request()

Prototype: Request_t get_command_request().
Arguments: None.
Returns: Request_t (enumerated type).
Description: Gets the received command descriptor to determine command request type. A command type may be REQ_READ or REQ_WRITE. These type values are defined in the enumerated type called Request_t which is imported with the package named avalon_mm_pkg.
get_response_queue_size()

Prototype: int get_response_queue_size().
Arguments: None.
Returns: int.
Description: Queries the response queue to determine number of response descriptors pending.

get_version()

Prototype: string get_version().
Arguments: None.
Returns: String.
Description: Return BFM version as a string of three integers separated by periods. For example, version 9.1 sp1 is encoded as '9.1.1'.

init()

Prototype: init().
Arguments: None.
Returns: void.
Description: Initializes the Avalon-MM slave interface.

pop_command()

Prototype: void pop_command().
Arguments: None.
Returns: void.
Description: Pops the command descriptor from the queue so that the testbench can query it using the get_command methods.

push_response()

Prototype: void push_response().
Arguments: None.
Returns: void.
Description: Pushes the fully populated response transaction descriptor onto the response queue. The BFM pops response descriptors off the queue as soon as they are available, reads them, and drives the Avalon-MM interface response plane.
set_command_transaction_mode()

Prototype: void set_command_transaction_mode (int mode);
Arguments: mode.
Returns: void.
Description: By default, write burst commands are consolidated into a single command transaction containing
the write data for all burst cycles in that command. This mode is set when the mode argument
equals 0. When the mode argument is set to 1, the default is overridden and write burst commands
yield one command transaction per burst cycle.

set_interface_wait_time()

Prototype: void set_interface_wait_time(int wait_cycles, int index);
Arguments: wait_cycles.
index.
Returns: void.
Description: Specifies zero or more wait states to assert in each Avalon burst cycle by driving waitrequest
active. With write burst commands, each write data cycle is forced to wait the number of cycles
corresponding to the cycle index. With read burst commands, there is only one command cycle
corresponding to index 0 which can be forced to wait.

set_response_burst_size()

Prototype: void set_response_burst_size(bit [AV_BURSTCOUNT_W-1:0]
burst_size).
Arguments: burst_size.
Returns: void.
Description: Sets the transaction burst count in the response descriptor.

set_response_data()

Arguments: data.
index.
Returns: void.
Description: Sets the transaction read data in the response descriptor. For burst transactions, the command
descriptor holds an array of data, with each element individually set by this method.
set_response_latency()

Arguments:    latency.
              index.
Returns:      void.
Description:  Sets the response latency for read commands. The response is driven out the specified number of cycles after receiving the read command.

set_response_timeout()

Prototype:     void set_response_timeout(int cycles).
Arguments:    None.
Returns:      void.
Description:  Set the number of cycles that may elapse before time out.

signal_command_received

Prototype:     signal_command_received.
Arguments:    None.
Returns:      void.
Description:  This event notifies the testbench that a command has been detected on an Avalon-MM port. The testbench can respond with a set_command_wait_time call on receiving this event to dynamically back pressure the driving Avalon-MM master. Alternatively, wait_time which was previously set may be used continuously for a set of transactions.

signal_error_exceed_max_pending_reads

Prototype:     signal_error_exceed_max_pending_reads.
Arguments:    None.
Returns:      void.
Description:  This event notifies the test bench of the error condition in which the slave has more than max_pending_reads pipelined read commands queued waiting to be processed.

signal_fatal_error

Prototype:     signal_fatal_error.
Arguments:    None.
Returns:      void.
Description:  This event notifies the test bench that a fatal error has occurred in this module.
The Avalon-MM Slave BFM with Avalon-ST API Wrapper provides VHDL support for the Avalon-MM Slave BFM. You can use the Avalon-MM Slave BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation. This component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-MM BFM’s method calls and returns into signals that are carried on the call and return interface ports. To call a method, the method identifier is pushed into the wrapper component via the channel field; the data is the arguments for the method. Once the method is complete, the data field transports the arguments for the method call. The response is returned on the response Avalon ST interface, and that Avalon ST data signal carries the return value. The wrapper is necessary because VHDL can only access ports and does not support the method calls across hierarchical boundaries used in the Avalon-MM Master BFM field. Figure 5–1 provides a high-level view of this VHDL testbench communicating with the BFM.

**Figure 5–1.** Avalon-MM Slave BFM with Avalon-ST Wrapper
In Figure 5–1, the API call interface and Avalon-ST call and return interface operate in separate clock domains with \( av\_clk \) synchronizing the FPGA logic and \( api\_clk \) synchronizing the Avalon-ST translation interface. Typically, the Avalon-ST interface, which is not part of the actual hardware design operates at a much faster frequency than the Avalon-MM Slave BFM interface, enabling 1000 API calls and returns to be issued to the BFM per Avalon clock cycle.

For every function call in the BFM, there is a channel identifier which stores the fixed mapping between channel number and the function. 

\(<$install_dir>/ip/altera/sopc_builder_ip/verification/lib/altera_avalon_components_pkg.vhd\) defines the following function calls:

- \texttt{MM\_SLV\_SIGNAL\_FATAL\_ERROR}\n- \texttt{MM\_SLV\_SIGNAL\_ERROR\_EXCEED\_MAX\_PENDING\_READS}\n- \texttt{MM\_SLV\_SIGNAL\_COMMAND\_RECEIVED}\n- \texttt{MM\_SLV\_SIGNAL\_RESP\_ISSUED}\n- \texttt{MM\_SLV\_SIGNAL\_RESERVED\_4}\n- \texttt{MM\_SLV\_SIGNAL\_RESERVED\_5}\n- \texttt{MM\_SLV\_SIGNAL\_RESERVED\_6}\n- \texttt{MM\_SLV\_SIGNAL\_RESERVED\_7}

With the exception of the API wrapper which provides an alternative way to access the Avalon-MM Slave BFM API to support VHDL testbenches, this component is identical to the Avalon-MM Slave BFM. For more information about this component refer the Chapter 4, The Avalon Memory-Mapped Slave BFM.
The Avalon-MM Monitor verifies Avalon-MM interfaces using SystemVerilog assertions. In addition, it provides test coverage reports so that you can determine when your test vectors provide sufficient test coverage for your component’s functionality. Using the Avalon-MM Monitor has three key advantages:

- It provides independent verification of the Avalon-MM protocol.
- It simplifies the verification task by providing a key component of the verification environment.
- It provides test coverage statistics, so that you know when you have completed sufficient testing.

The Avalon-MM Monitor is implemented in SystemVerilog and uses the SystemVerilog Assertion (SVA) language. SVA is supported by the Synopsis VCS, and Mentor Questa simulators. If you are using ModelSim, the monitor component still compiles and simulates, but the assertion checking is turned off.

Figure 6–1 shows a testbench that uses an Avalon-MM Monitor to test components with Avalon-MM interfaces. As this figure illustrates, the monitor’s Avalon-MM Master interface is connected to a component’s Avalon-MM slave interface, and an Avalon-MM Slave interface is connected to a component’s Avalon-MM master interface. The test program communicates with the monitor. The test program can use the monitor’s assertion checking and coverage groups to assure that all legal parameter values for the DUT’s Avalon-MM interface are tested.

**Figure 6–1. Testbench Using with an Avalon-MM Monitor**
Parameters

The Avalon-MM Monitor supports the full range of signals defined for the Avalon-MM master and slave interfaces. You can customize the Avalon-MM master and slave interfaces using the parameters described in Table 6–1.

For detailed information about the Avalon-MM interface, including timing diagrams, refer to the Avalon Interface Specifications.

Table 6–1. Parameters for the Avalon-MM Monitor  (Part 1 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Widths</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address width</td>
<td>32</td>
<td></td>
<td>Address width in bits.</td>
</tr>
<tr>
<td>Symbol width</td>
<td>8</td>
<td></td>
<td>Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.</td>
</tr>
<tr>
<td>Number of symbols</td>
<td>4</td>
<td></td>
<td>Numbers of symbols per word.</td>
</tr>
<tr>
<td>Burstcount width</td>
<td>3</td>
<td></td>
<td>The width of the burst count in bits.</td>
</tr>
<tr>
<td><strong>Port Enables</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use the read signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a read pin.</td>
</tr>
<tr>
<td>Use the write signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a write pin.</td>
</tr>
<tr>
<td>Use the address signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes address pins.</td>
</tr>
<tr>
<td>Use the byte enable signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes byte_enable pins.</td>
</tr>
<tr>
<td>Use the burstcount signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes burstcount pins.</td>
</tr>
<tr>
<td>Use the readdata signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a readdata pin.</td>
</tr>
<tr>
<td>Use the readdatavalid signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a readdatavalid pin.</td>
</tr>
<tr>
<td>Use the writedata signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a writedata pin.</td>
</tr>
<tr>
<td>Use the begintransfer signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes a begintransfer signal.</td>
</tr>
<tr>
<td>Use the beginbursttransfer signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes a beginbursttransfer signal.</td>
</tr>
<tr>
<td>Use the waitrequest signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a waitrequest pin.</td>
</tr>
<tr>
<td><strong>Burst Attributes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linewrap burst</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the address for bursts wraps instead of an incrementing. With a wrapping burst, when the address reaches a burst boundary, it wraps back to the previous burst boundary such that only the low order bits need to be used for addressing.</td>
</tr>
<tr>
<td>Burst on burst boundaries only</td>
<td>On</td>
<td>On/Off</td>
<td>When On, memory bursts are aligned to the address size.</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read response timeout (cycles)</td>
<td>100</td>
<td></td>
<td>Specifies when a timeout will occur if readdatavalid is not asserted.</td>
</tr>
<tr>
<td>Avalon write timeout (cycles)</td>
<td>100</td>
<td></td>
<td>Specifies when a timeout will occur if a burst write transfer has not completed.</td>
</tr>
</tbody>
</table>
The Avalon-MM Monitor’s API includes 21 methods that allow you to control the assertion checking and functional coverage of Avalon-MM master and slave interfaces. For example, the enable_waitrequest_timeout method enables the logic that verifies that the waitrequest signal is asserted for fewer cycles than the waitrequest timeout period. If the timeout period is violated, an error message displays on the console running the simulation. Error flags are also displayed in the waveform viewer.

Coverage groups ensure that the verification suite tests all expected functionality of the interface. For example, the cover_b2b_read_write method ensures that the verification suite includes a test for sequential read and write commands. The Avalon-MM Monitor includes 16 coverage groups.

The methods in the API are used to enable or disable assertions and coverage groups. By default all assertions and coverage groups are enabled. However, depending on the parameterization of the Avalon-MM interface, some methods are automatically disabled. For example, if the interface does not allow burst transfers, the coverage groups that test burst transfers are automatically disabled. Occasionally, you may need to turn off some assertion checking to avoid generating error messages when injecting protocol errors to test the Avalon-MM component’s error handling capability.

The names of all methods that enable assertions begin with enable_a. The names of all methods that enable coverage functionality begin with enable_c.

By default, if your testbench includes the Avalon-MM monitor, the checking function is enabled. You can disable checking with the DISABLE_ALTERA_AVALON_SIM_SVA macro.

To generate the coverage report when using the Synopsys VCS simulator, you can execute the following command:

```
urgent -dir simv.vdb
```
enable_a_beginbursttransfer_exist()

Prototype: enable_a_beginbursttransfer_exist().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures `beginbursttransfer` is asserted during any single burst transfer. It is disabled when `beginbursttransfer` is not used.

enable_a_beginbursttransfer_legal()

Prototype: enable_a_beginbursttransfer_legal().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures `beginbursttransfer` is asserted with a read or write signal. It is disabled when `beginbursttransfer` is not used.

enable_a_beginbursttransfer_single_cycle()

Prototype: enable_a_beginbursttransfer_single_cycle().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures `beginbursttransfer` is asserted for a single cycle regardless of the behavior of the `waitrequest` signal. It is disabled when `beginbursttransfer` is not used.

enable_a_begintransfer_exist()

Prototype: enable_a_begintransfer_exist().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures `begintransfer` is asserted during any single transfer. Disabled when either `begintransfer` is not supported.

enable_a_begintransfer_legal()

Prototype: enable_a_begintransfer_legal().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures `begintransfer` is asserted together with either `read` or `write`. Disabled when either `begintransfer` is not supported.
enable_a_begintransfer_single_cycle()

Prototype: enable_a_begintransfer_single_cycle().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures \texttt{begintransfer} is asserted for only 1 cycle and not reasserted for any single transfer, regardless of the status of the \texttt{waitrequest} signal.

enable_a_burst_legal()

Prototype: enable_a_burst_legal().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that the total number of assertions for the \texttt{write} and \texttt{readdatavalid} is the same as the \texttt{burstcount} for any burst transfer. Disabled when burst transfers are not supported.

enable_a_byteenable_legal()

Prototype: enable_a_byteenable_legal().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures the \texttt{byteenable} value is legal value as specified by the \texttt{Avalon Interface Specifications}. Disabled when \texttt{byteenable} is not supported.

enable_a_constant_during_burst()

Prototype: enable_a_constant_during_burst().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that \texttt{address} and \texttt{burstcount}, and \texttt{byteenable} are held constant if a write burst transfer. Disabled when \texttt{waitrequest} is not supported. It is disabled when burst transfers are not supported.

enable_a_constant_during_waitrequest()

Prototype: enable_a_constant_during_waitrequest().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that \texttt{read}, \texttt{write}, \texttt{writedata}, \texttt{address}, \texttt{burstcount}, and \texttt{byteenable} are held constant if \texttt{waitrequest} is asserted. Disabled when \texttt{waitrequest} is not supported.
enable_a_exclusive_read_write()

Prototype: enable_a_exclusive_read_write().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures read and write are not asserted simultaneously. Disabled when either read or write is not supported.

enable_a_half_cycle_reset_legal()

Prototype: enable_a_half_cycle_reset_legal().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures reset is asserted correctly.

enable_a_less_than_burstcount_max_size()

Prototype: enable_a_less_than_burstcount_max_size().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures burstcount size is less than or equal to maxBurstSize. It is disabled when either burst transfers are not supported or the bust size is less than 1.

enable_a_less_than_maximumpendingreadtransactions()

Prototype: enable_a_less_than_maximumpendingreadtransactions().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that the number of pending read transfers is less than maximumPendingReadTransactions. Disabled when either read is not supported or maximumPendingReadTransactions is less than 1.

enable_a_no_readdatavalid_during_reset()

Prototype: enable_a_no_readdatavalid_during_reset().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that readdatavalid is deasserted if reset is asserted. Disabled when readdatavalid is not supported.
enable_a_read_response_timeout()

Prototype: enable_a_read_response_timeout()
Arguments: Boolean.
Returns: Void.
Description: This assertion ensures readdatavalid is asserted within maximum allowed timeout period. Disabled when either readdatavalid is not supported or the maximum allowed timeout period is less than 1.

enable_a_no_read_during_reset()

Prototype: enable_a_no_read_during_reset()
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures read is deasserted if reset is asserted. Disabled when read is not supported.

enable_a_no_write_during_reset()

Prototype: enable_a_no_write_during_reset()
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures write is deasserted if reset is asserted. Disabled when write is not supported.

enable_a_waitrequest_during_reset()

Prototype: enable_a_waitrequest_during_reset()
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that waitrequest is asserted if reset is asserted. Disabled when waitrequest is not supported.

enable_a_waitrequest_timeout()

Prototype: enable_a_waitrequest_timeout()
Arguments: Boolean.
Returns: Void.
Description: This assertion ensures waitrequest is not asserted continuously for more than maximum allowed timeout period. Disabled when either waitrequest is not supported or the maximum timeout period is less than 1.
enable_a_writeburst_timeout()

Prototype: enable_a_writeburst_timeout().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that the write burst transfer is completed within maximum allowed timeout period. Disabled when either write burst transfers are not supported or the write burst timeout period is less than 1 cycle.

enable_c_b2b_read_read()

Prototype: enable_c_b2b_read_read().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test back-to-back read transfers. This method is disabled when reads are not supported.

enable_c_b2b_read_write()

Prototype: enable_c_b2b_read_write().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test a read transfer immediately followed by a write transfer. This method is disabled when reads or writes are not supported.

enable_c_b2b_write_read()

Prototype: enable_c_b2b_write_read().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test a write transfer immediately followed by a read. This method is disabled if either reads or writes are not supported.

enable_c_b2b_write_write()

Prototype: enable_c_b2b_write_write().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test back-to-back write transfers. This method is disabled if writes are not supported.
enable_c_idle_in_read_response()

Prototype: void enable_c_idle_in_read_response().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to count idle cycles during a read burst response. This method is disabled if reads or readdatavalids are not supported.

enable_c_idle_in_write_burst()

Prototype: void enable_c_idle_in_write_burst().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to count idle cycles during a write burst transaction. This method is disabled if writes are not supported.

enable_c_pending_read()

Prototype: void enable_c_pending_read().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test pending read support. It covers all values for up to the maximum number of pending reads. This method is disabled when either reads or pipelined reads are not supported.

enable_c_read()

Prototype: void enable_c_read().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test read transfers. This method is disabled when reads are not supported.

enable_c_read_burstcount()

Prototype: void enable_c_read_burstcount().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group tests different sizes of burstcount during read burst transfers. It tests all possible values of burstcount. This method is disabled when either burst transfers or reads are not supported, or the maximum burst is less than 1.
enable_c_read_byteenable()

Prototype: enable_c_read_byteenable()
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to ensure all legal values of the `byteenable` signal are asserted during read transfers. It is disabled when either `byteenable` or `read` is not supported.

enable_c_read_latency()

Prototype: enable_c_read_latency()
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test all values of the read latency parameter. This method is disabled if `read` or `readdatavalids` are not supported, or if the maximum read latency is less than 1.

enable_c_waitrequested_read()

Prototype: enable_c_waitrequested_read()
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test all values of the wait request timeout parameter during read transfers. This method is disabled if `read` or `waitrequest` are not supported, or if the wait request timeout period is less than 1.

enable_c_waitrequested_write()

 Prototype: enable_c_waitrequested_write()
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test all values of the wait request timeout parameter. This method is disabled if `write` or `waitrequest` are not supported, or if the wait request timeout period is less than 1.

enable_c_write()

 Prototype: enable_c_write()
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage group to test write transfers. This method is disabled when writes are not supported.
<table>
<thead>
<tr>
<th>Function</th>
<th>Prototype</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable_c_write_burstcount</td>
<td>enable_c_write_burstcount()</td>
<td>Boolean</td>
<td>Void</td>
<td>This method enables a coverage group to test different sizes of <code>burstcount</code> during write burst transfers. It tests all possible values of <code>burstcount</code>. This method is disabled when either burst transfers or writes are not supported, or the maximum burst is less than 1.</td>
</tr>
<tr>
<td>enable_c_write_byteenable</td>
<td>enable_c_write_byteenable()</td>
<td>Boolean</td>
<td>Void</td>
<td>This method enables a coverage group ensures all legal values of the <code>byteenable</code> signal are asserted during write transfers. It is disabled when either <code>byteenable</code> or <code>write</code> is not supported.</td>
</tr>
</tbody>
</table>
This chapter demonstrates how to use the Avalon Memory-Mapped (Avalon-MM) master and slave BFMs to verify Avalon-MM master and slave components in an SOPC Builder design. In the first example, an on-chip RAM which includes an Avalon-MM slave port, is the DUT. Its behavior is verified using the Avalon-MM master BFM component. The second example verifies an Avalon-MM master DUT using the Avalon-MM slave BFM component. Both examples cover the following topics:

- Overview of the Test Setup
- Setting Up the Test
- Running the Simulation
- Observing the Results

### Overview of the Test Setup—Avalon-MM Slave DUT

Figure 7–1 illustrates the top-level testbench to verify an Avalon-MM slave component. An on-chip RAM component is connected to the Avalon-MM Master BFM in SOPC Builder. The test program initializes the Avalon-MM master BFM. Once initialization and system reset have completed, the test program instructs the master BFM to write random data to the slave DUT. The write data is also saved into a local array for future reference. The master BFM reads back the data written, compares it to the data stored in the local array, and reports mismatches. The test passes if all the read data is correct.

Example 7–1, excerpted from `master_bfm_tb.v`, demonstrates the use of the Avalon-MM Master API. It shows the following two tasks:

- `master_set_and_push_commands`—This task sets the fields of the command descriptor and pushes it on to the command queue.
- `master_pop_and_get_response`—This task pops the response received by the master BFM.
As these tasks illustrates, you use the `set_command_<field>` methods to define the command and the `push_command` method to add it to the queue. You use the `pop_response` method to get a response and the `get_response_<field>` to retrieve the fields of the response.

**Example 7–1. Verilog Tasks Illustrating the Avalon-MM Master BFM API**

```verilog
//this task sets the command descriptor for master BFM and push it to the queue
// . . .
begin
`MSTR_BFM.set_command_request(request);
`MSTR_BFM.set_command_address(addr);
`MSTR_BFM.set_command_byte_enable(byte_enable, `INDEX_ZERO);
`MSTR_BFM.set_command_idle(idle, `INDEX_ZERO);
`MSTR_BFM.set_command_init_latency(init_latency);

if (request == REQ_WRITE)
begin
`MSTR_BFM.set_command_data(data, `INDEX_ZERO);
end

`MSTR_BFM.push_command();
end
endtask

//this task pops the response received by master BFM from queue
// . . .
begin
`MSTR_BFM.pop_response();
request = Request_t' (`MSTR_BFM.get_response_request());
addr = `MSTR_BFM.get_response_address();
data = `MSTR_BFM.get_response_data(`INDEX_ZERO);
end
endtask
```

For more information on the methods used by the Avalon-MM Master BFM, refer to the “Application Program Interface (API)” on page 2–9 in the Avalon Memory-Mapped Master BFM.

Although this testbench is written in Verilog HDL, the Avalon Verification IP Suite supports VHDL by providing wrappers for the Avalon-MM Master and Slave BFMS. You can include the BFMs with wrappers in simulators that support mixed language simulation. For more information, refer to Chapter 3, Avalon Memory-Mapped Master BFM with Avalon-ST API Wrapper and Chapter 5, Avalon Memory-Mapped Slave BFM with Avalon-ST API Wrapper.

**Setting Up the Test**

This section describes the steps to verify the on-chip RAM using the Avalon-MM master BFM. Running this test requires the following software and files:

- Quartus II software, version 9.1.
- ModelSim-AE software. This tutorial was run with ModelSim-AE 6.5b which is available for download with the Quartus II 9.1 software.
The `avalon_mm_bfm_ex.zip` file. This design file is included with the *Avalon Verification IP Suite User Guide* on the Altera website.

In order to run the example test, unzip the `avalon_mm_bfm_ex.zip` file to a working directory on your hard drive. This location is referred to as `<working_directory>`.

1. Choose **Programs > Altera > Quartus II><version number>** (Windows Start menu) to run the Quartus II software.

2. Open the `master_bfm_project.qpf` file located in `<working_directory>\avalon_mm_bfm_ex\tutorial_master_bfm`.

3. To open SOPC, on the Tools menu, click **SOPC Builder**.

4. On the **System Contents** tab, under **Verification Suite**, double-click the **Avalon MM Master BFM**.

5. In the configuration wizard, change the parameter values to match those listed in Table 7–1.

### Table 7–1. Master BFM Parameter Values  (Part 1 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Widths</strong></td>
<td></td>
</tr>
<tr>
<td>Address width</td>
<td>16</td>
</tr>
<tr>
<td>Symbol width</td>
<td>8</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>Number of symbols</td>
<td>4</td>
</tr>
<tr>
<td>Burstcount width</td>
<td>3</td>
</tr>
<tr>
<td><strong>Port Enables</strong></td>
<td></td>
</tr>
<tr>
<td>Use the read signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the write signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the address signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the byteenable signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the burstcount signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the readdata signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the readdatavalid signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the writedata signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the begintransfer signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the beginbursttransfer signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the waitrequest signal</td>
<td>On</td>
</tr>
<tr>
<td><strong>Port Polarity</strong></td>
<td></td>
</tr>
<tr>
<td>Assert reset high</td>
<td>On</td>
</tr>
<tr>
<td>Assert waitrequest high</td>
<td>On</td>
</tr>
<tr>
<td>Assert read high</td>
<td>On</td>
</tr>
<tr>
<td>Assert write high</td>
<td>On</td>
</tr>
<tr>
<td>Assert byteenable high</td>
<td>On</td>
</tr>
</tbody>
</table>
Table 7–1. Master BFM Parameter Values (Part 2 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert readdatavalid high</td>
<td>On</td>
</tr>
<tr>
<td>Burst Attributes</td>
<td></td>
</tr>
<tr>
<td>Linewrap burst</td>
<td>Off</td>
</tr>
<tr>
<td>Burst on burst boundaries only</td>
<td>Off</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td></td>
</tr>
<tr>
<td>Maximum pending reads</td>
<td>1</td>
</tr>
<tr>
<td>Fixed read latency (cycles)</td>
<td>0</td>
</tr>
</tbody>
</table>

6. Click Finish.

7. Right-click on the component and select Rename. Change the component name to master_bfm.

8. In the System Contents tab, double-click the On-Chip Memory (RAM or ROM) in the On-Chip subfolder of the Memories and Memory Controllers folder.

9. Retain the default settings for the on-chip RAM and click Finish.

10. Right-click on the RAM and click Rename. Change the component name to ram.

11. Connect the master_bfm m0 Avalon-MM master port to the onchip_mem s1 Avalon-MM slave port using the following procedure:
   a. Click on the m0 port then hover in the Connections column to display possible connections.
   b. Click on the open dot at the intersection of the onchip_mem s1 port and the master_bfm m0 to create a connection.

12. Click Generate. Save the system if you are prompted to do so.

**Running the Simulation**

To run the simulation, complete the following steps:

1. Start the ModelSim software.
3. Navigate to <working_directory>\avalon_mm_bfm_ex\tutorial_master_bfm and click OK.
4. On the Compile menu, click Compile Options.
5. Click the Verilog & SystemVerilog tab.
6. In the Language Syntax box, select Use SystemVerilog and click OK.
7. On the File menu, point to New and click Library.
8. In the Create a New Library dialog box, select a new library and a logical mapping to it.
9. Type work in both Library Name and Library Physical Name box and click OK.
10. On the Compile menu, click Compile.
11. In the Compile Source Files dialog box, select `master_bfm_sopc.v` and click Compile. Next, click `master_bfm_tb.v` and click Compile.

12. When all the compilations are complete, click Done.

13. To start the simulation, on the Simulate menu, click Start Stimulation.

14. In the Start Simulation dialog box, expand the work directory.

15. Select `master_bfm_tb` and click OK.

16. To open a wave viewer, on the View menu, click Wave.

17. To load a default set of simulation signals into the wave viewer, on the File menu, click Load.

18. In the Open Format dialog box, select the `wave.do` file and click Open.

19. To run the simulation, in the Transcript window, type run 500 ns

If you are running ModelSim-SE you need to use the –novopt option to prevent ModelSim from optimizing the design, making the signals specified for the wave viewer unavailable. You can use the script provided with this example instead of completing Steps 7–19, type the following command in the Transcript window:

```
do script.do
```

### Observing the Results

In this test, the Avalon-MM master BFM writes five words of random data to the on-chip memory (DUT). The master BFM then reads back the five words and compares the data read to the expected values. If simulation is successful, the message shown in Example 7–2 appears.

**Example 7–2.** Message in ModelSim Transcript when Running Simulation for Avalon-MM Slave DUT

```
960000: INFO: master_bfm_tb: Test has completed. 5 pass, 0 fail
```

Figure 7–2 shows the waveform when the Avalon-MM master BFM for the writes and reads to the slave DUT.
Overview of the Test Setup—Avalon-MM Master DUT

Figure 7–3 illustrates the top-level testbench to verify an Avalon-MM master component using an Avalon-MM slave BFM. The Avalon-MM master DUT is a simple write-read master which it writes data to a slave component and reads back the data written.

The amount of data written is specified by the master’s BLOCKSIZE parameter. The default value for this parameter is 4. When all data is written, the master DUT reads the data back from slave BFM and checks it against the expected data. If a mismatch occurs, the master DUT asserts its exported error signal.
The slave BFM component responds to master’s commands when the signal_command_received event is triggered. The test program pops the master command out of slave BFM component’s client command queue. If the command is a write, the write data is saved to a local array. For read commands, data is read out of the local array. The test program then constructs a response descriptor with the read data. The slave BFM drives the response to the master DUT. The test ends after the master DUT has received all responses from slave BFM. The test passes if the master DUT does not assert its error signal.

For more information on the methods used by the Avalon-MM slave BFM to construct commands, refer to the “Application Program Interface (API)” on page 4–8 of the Avalon Memory-Mapped Slave BFM.

Setting Up the Test

This section describes the steps to verify the Avalon-MM master using the Avalon-MM slave BFM. You must have the avalon_mm_bfm_ex.zip file extracted to your local drive before you proceed, as described previously in “Setting Up the Test” on page 7–2.

To run this test, complete this following steps:

1. Open the slave_bfm_project.qpf file located in 
   <working_directory>\avalon_mm_bfm_ex\tutorial_slave_bfm.
2. On the Tools menu, click SOPC Builder.
3. To create the design, in the System Contents tab, expand BFM Tutorial and double-click Write-Read Master.
4. Retrain the default values given in the configuration wizard and click Finish to add this component to your system.
5. Right-click on the component and click Rename. Change the component name to master.
7. Double-click Altera Avalon MM Slave BFM.
8. In the configuration wizard, change the parameter values to match those listed in Table 7–2.

Table 7–2. Avalon-MM Slave BFM Parameter Values (Part 1 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Widths</strong></td>
<td></td>
</tr>
<tr>
<td>Address width</td>
<td>16</td>
</tr>
<tr>
<td>Symbol width</td>
<td>8</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>Number of symbols</td>
<td>4</td>
</tr>
<tr>
<td>Burstcount width</td>
<td>3</td>
</tr>
<tr>
<td><strong>Port Enables</strong></td>
<td></td>
</tr>
</tbody>
</table>
9. Click **Finish**.

10. Right-click on the component and select **Rename**. Change the component name to `slave_bfm`.

11. Connect the master `m0_avalon_master` port to the slave_bfm `s0` Avalon slave port using the following procedure:
   a. Click on the master `m0` port then hover in the **Connections** column to display possible connections.
   b. Click on the open dot at the intersection of the slave_bfm `s0` port and the `m0` port to create a connection.

12. Click **Generate**. Save the system if you are prompted to do so.

### Table 7–2. Avalon-MM Slave BFM Parameter Values (Part 2 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the read signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the write signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the address signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the byte enable signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the burstcount signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the readdata signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the readdatavalid signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the writedata signal</td>
<td>On</td>
</tr>
<tr>
<td>Use the begintransfer signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the beginbursttransfer signal</td>
<td>Off</td>
</tr>
<tr>
<td>Use the waitrequest signal</td>
<td>On</td>
</tr>
</tbody>
</table>

**Port Polarity**

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert reset high</td>
<td>On</td>
</tr>
<tr>
<td>Assert waitrequest high</td>
<td>On</td>
</tr>
<tr>
<td>Assert read high</td>
<td>On</td>
</tr>
<tr>
<td>Assert write high</td>
<td>On</td>
</tr>
<tr>
<td>Assert byteenable high</td>
<td>On</td>
</tr>
<tr>
<td>Assert readdatavalid high</td>
<td>On</td>
</tr>
</tbody>
</table>

**Burst Attributes**

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linewrap burst</td>
<td>Off</td>
</tr>
<tr>
<td>Burst on burst boundaries only</td>
<td>Off</td>
</tr>
</tbody>
</table>

**Miscellaneous**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum pending reads</td>
<td>1</td>
</tr>
<tr>
<td>Fixed read latency (cycles)</td>
<td>0</td>
</tr>
<tr>
<td>Fixed read wait time (cycles)</td>
<td>1</td>
</tr>
<tr>
<td>Fixed write wait time (cycles)</td>
<td>0</td>
</tr>
</tbody>
</table>
Running the Simulation

1. Start the ModelSim software.
3. Navigate to `<working_directory>/avalon_mm_bfm_ex/tutorial_slave_bfm` and click OK.
4. On the Compile menu, click Compile Options.
5. Click the Select Verilog & SystemVerilog tab.
6. In the Language Syntax box, select Use SystemVerilog and click OK.
7. On the File menu, point to New and click Library.
8. In the Create a New Library dialog box, click a new library and a logical mapping to it.
9. Type `work` in both Library Name and Library Physical Name boxes and click OK.
10. On the Compile menu, click Compile.
11. In the Compile Source Files dialog box, select `slave_bfm_sopc.v` and click Compile. Next, click `slave_bfm_tb.v` and click Compile.
12. Click Done when all the compilations are completed.
14. In the Start Simulation dialog box, expand the work category.
15. Select `slave_bfm_tb` and click OK.
16. To open a waveform viewer, on the View menu, click Wave.
17. To load a default set of import signals, on the File menu, click Load.
18. In the Open Format dialog box, select `wave.do` and click Open.
19. To run the simulation, in the ModelSim console window, type `run 600 ns`.

If you are running ModelSim-SE you need to use the `-novopt` option to prevent ModelSim from optimizing the design, making the signals specified in for the waveform viewer unavailable. You can use the script provided with this example instead of completing Steps 7–19, type the following command in the Transcript window:

```bash
do script.do
```

Observing the Results

In this example test, the master DUT writes four data words to the slave BFM component and reads them back. The test program displays messages in the ModelSim console window every time the slave BFM component receives master command. Example 7–3 shows a partial transcript from a successful run.
Example 7–3. Message in ModelSim Transcript when Running Simulation for Avalon-MM Master DUT

```
# 251000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: write addr 0
# 251000: INFO: slave_bfm_tb: Master Write request to address 0000 with data 00000000
# 291000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: write addr 1
# 291000: INFO: slave_bfm_tb: Master Write request to address 0001 with data 00000002
# 331000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: write addr 2
# 331000: INFO: slave_bfm_tb: Master Write request to address 0002 with data 00000006
# 371000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: write addr 3
# 371000: INFO: slave_bfm_tb: Master Write request to address 0003 with data 0000000e
# 411000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: read addr 0
# 411000: INFO: slave_bfm_tb: Master Read request from address 0000
# 411000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.push_response: push
# response: read addr 0
# 451000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: read addr 1
# 451000: INFO: slave_bfm_tb: Master Read request from address 0001
# 451000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.push_response: push
# response: read addr 1
# 491000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: read addr 2
# 491000: INFO: slave_bfm_tb: Master Read request from address 0002
# 491000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.push_response: push
# response: read addr 2
# 531000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.pop_command: read addr 3
# 531000: INFO: slave_bfm_tb: Master Read request from address 0003
# 531000: INFO: slave_bfm_tb.tb.DUT.the_slave_bfm.slave_bfm.push_response: push
# response: read addr 3
```

Figure 7–4 shows the waveforms for the Avalon-MM master DUT write and reads to the Avalon-MM slave BFM component.

Figure 7–4. Avalon-MM Master Writes and Reads to Avalon-MM Slave BFM

---

Avalon Verification IP Suite User Guide © November 2009 Altera Corporation
The Avalon-ST Source BFM implements the Avalon-ST interface protocol, a protocol that is point-to-point, packet oriented, and drives unidirectional data. This BFM component includes a procedural interface to control signals on the Avalon-ST interface, including: ready, start of packet, and end of packet. Figure 8–1 shows the top-level modules for testbench that uses the Avalon-ST Source BFM to verify an Avalon-ST sink component. In addition to the Altera-provided Avalon-ST Source BFM component, the testbench typically includes a test program and the device under test (DUT). Using the Altera-provided Avalon-ST Source BFM has two significant advantages:

- It accelerates development by supplying one of the modules for the verification testbench.
- It provides an independent check on the DUT implementation of the Avalon-ST protocol. Because the testbench includes an Avalon-ST Source designed by Altera and an Avalon-ST sink designed by the user, it should highlight any misinterpretation of the protocol implemented by the DUT which might be missed in a testbench designed by a single engineer.

The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitors components verify protocol compliance.

Figure 8–1. Top-Level Module to Verify an Avalon-ST Sink Device

For more information about the Avalon-ST specification refer to the Avalon Interface Specifications.

**Functional Description**

This section provides a functional description of the Avalon-ST Source BFM. It includes the following topics:

“Timing” on page 8–2

“Block Diagram” on page 8–2
Timing

The timing diagram shown in Figure 8–2 illustrates the timing for an Avalon-ST Source BFM sending data to a sink. In the first instance the sink is not ready when the source has data. In the second instance, the sink is ready but the source does not initially have valid data.

The Avalon-ST BFM behaves differently depending on whether the sink’s READY_LATENCY = 0 or READY_LATENCY > 0. When the ready latency is 0, the source BFM holds its current transaction until the sink is ready. When the ready latency is greater than 0, the BFM drives idles until the sink is ready, then it drives the transaction. Figure 8–2 illustrates the timing when READY_LATENCY = 0.

Figure 8–2. Avalon-ST Source Sending Data to a Sink

Table 8–1 explains the annotations used in Figure 8–2.

Table 8–1. Key to Annotations in Figure 8–2

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_idle</td>
<td>The idle time before a transaction. This time is set by the command set_transaction_idles.</td>
</tr>
<tr>
<td>T_{rl_1}</td>
<td>The response latency for the first source to sink transaction which is 3 cycles. The source gets this time using the get_response_latency command.</td>
</tr>
<tr>
<td>S_{src_dr}</td>
<td>Signals that the source is driving valid data. The event name is signal_src_driving_transaction.</td>
</tr>
<tr>
<td>S_{src_rdy}</td>
<td>Signals the source has received the assertion of ready from the sink. The event name is signal_src_ready.</td>
</tr>
<tr>
<td>S_{tc}</td>
<td>Signals the first transaction is complete. The event name is signal_src_transaction_complete.</td>
</tr>
<tr>
<td>S_{src_rdy}</td>
<td>Signals the source has received the deassertion of ready from the sink. The event name is signal_src_not_ready.</td>
</tr>
</tbody>
</table>

Block Diagram

Figure 8–3 provides a block diagram of the Avalon-ST Source BFM. As this figure illustrates, the BFM includes the following six major blocks.
Parameters

The Avalon-ST Source BFM supports all of the signals defined for the Avalon-MM source interface. You can customize the Avalon-ST Source interface using the parameters described in Table 8–2.

For detailed information about the Avalon-ST interface, including timing diagrams, refer to the *Avalon Interface Specifications.*
The API provides methods for a testbench to control and query this BFM component. Test programs must only use these public access methods and events to communicate with this BFM component.

The API includes methods fall into the three categories:

- Methods to initialize the BFM and determine its state
- Methods to create Avalon-ST transactions
- Methods to determine the response of the DUT

The following section presents these methods in alphabetical order.

Table 8–2. Parameters for the Avalon-ST Source BFM

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Enables</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Include the signals to support</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes the startofpacket, endofpacket, and empty</td>
</tr>
<tr>
<td>packets</td>
<td></td>
<td></td>
<td>signals.</td>
</tr>
<tr>
<td>Use the channel signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes channel pin or pins.</td>
</tr>
<tr>
<td>Use the error signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes error pin or pins.</td>
</tr>
<tr>
<td>Use the ready signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a ready pin.</td>
</tr>
<tr>
<td>Use the valid signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a valid pin.</td>
</tr>
<tr>
<td>Use the empty signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes empty pins.</td>
</tr>
<tr>
<td><strong>Port Widths</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbol Width</td>
<td>8</td>
<td></td>
<td>Data symbol width in bits. The symbol width should be 8 for byte-oriented</td>
</tr>
<tr>
<td>Number of symbols</td>
<td>4</td>
<td></td>
<td>Specifies the number of symbols that are transferred per beat.</td>
</tr>
<tr>
<td>Width of the channel signal</td>
<td>1</td>
<td></td>
<td>Specifies the width of the channel signal.</td>
</tr>
<tr>
<td>Width of the error signal</td>
<td>1</td>
<td></td>
<td>Specifies the width of the error signal.</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready latency</td>
<td>0</td>
<td></td>
<td>Specifies the delay between the ready and valid signals. Refer to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Avalon Interface Specification for more information.</td>
</tr>
<tr>
<td>Max channel number</td>
<td>1</td>
<td></td>
<td>Specifies the maximum number of channels that the interface supports.</td>
</tr>
<tr>
<td><strong>API Streaming Interface</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width of API interface data signal</td>
<td>64</td>
<td></td>
<td>The width of the data signal.</td>
</tr>
<tr>
<td>Width of API return interface data signal</td>
<td>64</td>
<td>The width of the return interface data signal.</td>
<td></td>
</tr>
</tbody>
</table>

Note to Table 8–2:
(1) This interface is only required for the Avalon-ST Source BFM with Avalon-ST API Wrapper which is used in mixed language simulations.
**get_response_latency()**

Prototype: `get_response_latency()`.
Arguments: None.
Returns: `int`.
Description: Returns the response latency in cycles due to back pressure for the most recently popped transaction.

**get_response_queue_size()**

Prototype: `get_response_queue_size()`.
Arguments: None.
Returns: `int`.
Description: Returns the number of transactions in the response queues.

**get_src_ready()**

Prototype: `get_src_ready()`.
Arguments: None.
Returns: `bit`.
Description: This method returns the value of the source ready port.

**get_src_transaction_complete()**

Prototype: `get_src_transaction_complete()`.
Arguments: None.
Returns: `bit`.
Description: This method returns the transaction complete status.

**get_transaction_queue_size()**

Prototype: `get_transaction_queue_size()`.
Arguments: None.
Returns: `int`.
Description: Returns the number of transactions in the local queues.

**get_version()**

Prototype: `get_version()`.
Arguments: None.
Returns: `String`.
Description: This method returns BFM version as a string of three integers separated by periods. For example, version 9.1 SP1 is encoded as “9.1.1”. 
**init()**

Prototype: `init()`.
Arguments: None.
Returns: `void`.
Description: Drives the interface to the idle state.

**pop_response()**

Prototype: `pop_response()`.
Arguments: None.
Returns: `void`.
Description: Pops the response transaction from the queue before querying contents.

**push_transaction()**

Prototype: `push_transaction()`.
Arguments: None.
Returns: `void`.
Description: Pushes the out-going transaction into the local transaction queue. The BFM drives the appropriate signals to the Avalon-ST interface based on the transactions in its local queue.

**set_response_timeout()**

Prototype: `set_response_timeout(int cycles)`.
Arguments: cycles—Sets the number of cycles that may elapse during backpressure before the time-out error is asserted. Disable the time-out by setting the cycles argument to zero.
Returns: `void`.
Description: This method sets the number of cycles that have to elapse before a response timeout is asserted.

**set_transaction_channel()**

Prototype: `set_transaction_channel(STChannel_t channel)`.
Arguments: channel.
Returns: `void`.
Description: Sets the channel identifier in the out-going transaction.

**set_transaction_data()**

Prototype: `set_transaction_data(STData_t data)`.
Arguments: data.
Returns: `void`.
Description: Sets the value of data in the out-going transaction.
set_transaction_idles()

Arguments: idle_cycles.
Returns: void.
Description: Sets the number of idle cycles to elapse before driving the out-going transaction.

set_transaction_eop()

Prototype: set_transaction_eop(bit eop).
Arguments: eop.
Returns: void.
Description: Sets the status of the end of packet signal in the out-going transaction.

set_transaction_empty()

Prototype: set_transaction_empty(STEmpty_t empty).
Arguments: empty.
Returns: void.
Description: Sets the out-going transaction empty value.

set_transaction_error()

Prototype: set_transaction_error(STError_t error).
Arguments: error.
Returns: void.
Description: Sets the out-going transaction error value.

set_transaction_sop()

Prototype: set_transaction_sop(bit sop).
Arguments: sop.
Returns: void.
Description: Sets the status of the start of packet signal in the out-going transaction.

signal_fatal_error

Prototype: signal_fatal_error.
Arguments: None.
Returns: void.
Description: This event signals that the a fatal error has occurred. It terminates the simulation.
signal_response_done

Prototype: signal_response_done.
Arguments: None.
Returns: void.
Description: This event signals that the response to a driven data beat is available.

signal_src_driving_transaction

Prototype: signal_src_driving_transaction.
Arguments: None.
Returns: void.
Description: This event signals when the source begins to drive a transaction to the interface.

signal_src_not_ready

Prototype: signal_src_not_ready.
Arguments: None.
Returns: void.
Description: This event signals that the ready signal is not asserted.

signal_src_ready

Prototype: signal_src_ready.
Arguments: None.
Returns: void.
Description: This event signals that the ready signal is asserted.

signal_src_transaction_complete

Prototype: signal_src_transaction_complete.
Arguments: None.
Returns: void.
Description: This event signals that all pending transactions have completed.
The Avalon-ST Source BFM with Avalon-ST API Wrapper provides VHDL support for the Avalon-ST Source BFM. You can use the Avalon-ST Source BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation. This component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-ST Source BFM’s method calls and returns into signals that are carried on the call and return interface ports. The wrapper is necessary because VHDL can only access ports and does not support the method calls used in the Avalon-ST Source BFM. Figure 9–1 provides a high-level view of this component.

In Figure 9–1, the API call interface and Avalon-ST call and return interface operate in separate clock domains with \texttt{av\_clk} synchronizing the FPGA logic and \texttt{api\_clk} synchronizing the Avalon-ST translation interface. Typically, the Avalon-ST interface, which is not part of the actual hardware design, operates at a much faster clock frequency than the Avalon-ST Source BFM interface.
For every function call in the BFM, there is a channel identifier which stores the fixed mapping between channel number and the function.

<install_dir>/ip/altera/sopc_builder_ip/verification/lib/altera_avalon_components_pkg.vhd defines the following function calls:

- ST_SRC_INIT
- ST_SRC_SET_RESP_TIMEOUT
- ST_SRC_PUSH_TRANS
- ST_SRC_GET_TRANS_QUEUE_SIZE
- ST_SRC_GET_RESP_QUEUE_SIZE
- ST_SRC_SET_TRANS_DATA
- ST_SRC_SET_TRANS_CHANNEL
- ST_SRC_SET_TRANS_IDLES
- ST_SRC_SET_TRANS_SOP
- ST_SRC_SET_TRANS_EOP
- ST_SRC_SET_TRANS_ERROR
- ST_SRC_SET_TRANSEMPTY
- ST_SRC_POP_RESP
- ST_SRC_GET_RESP_LATENCY
- ST_SRC_GET_SRC_READY
- ST_SRC_GET_SRC_TRANS_COMPLETE

With the exception of the API wrapper, this component is identical to the Avalon-ST Source BFM. For more information about this component refer the Chapter 8, Avalon Streaming Source BFM.
The Avalon-ST Sink BFM implements the Avalon-ST interface protocol, a protocol that is point-to-point, packet oriented, and drives unidirectional data. This BFM component also includes a procedural interface to respond to the device under test (DUT) that includes an Avalon-ST source interface. Figure 10–1 shows the top-level modules for testbench that uses the Avalon-ST Sink BFM to verify an Avalon-ST source device. In addition to the Altera-provided Avalon-ST Sink BFM component, the testbench includes a test program and the DUT. Using the Altera-provided Avalon-ST Sink BFM has two significant advantages:

- It accelerates development by supplying one of the modules for the verification testbench.
- It provides an independent check on the DUT implementation of the Avalon-ST protocol. Because the testbench includes an Avalon-ST Sink designed by Altera and an Avalon-ST sink designed by the user, it should highlight any misinterpretation of the protocol implemented by the DUT which might be missed in a testbench designed by a single engineer.

The BFMs allow illegal transactions so that you can test the error-handling functionality of your DUT; consequently, the BFMs cannot be relied upon to guarantee protocol compliance. The Avalon Monitors components verify protocol compliance.

For more information about the Avalon-ST specification refer to the Avalon Interface Specifications.

**Functional Description**

This section provides a functional description of the Avalon-ST Sink BFM. It includes the following topics:

“Timing” on page 10–2

“Block Diagram” on page 10–2
Timing

The timing diagram shown in Figure 10–2 illustrates the timing for an Avalon-ST Sink BFM signalling when it is ready to receive data from an Avalon-ST source. In the first instance the sink is not ready when the source has data. In the second instance, the sink is ready but the source does not initially have valid data.

Figure 10–2. Avalon-ST Source and Sink Timing

![Timing Diagram](image)

Table 10–1 explains the annotations used in Figure 10–2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_idle</td>
<td>The idle time between transactions. This time is reported by the command <code>get_transaction_idles</code>.</td>
</tr>
<tr>
<td>S_snk_rdyA</td>
<td>Signals the sink has asserted <code>ready</code>. The event name is <code>signal_snk_ready_assert</code>.</td>
</tr>
<tr>
<td>S_tr</td>
<td>Signals the transaction has been received and queued. The event name is <code>signal_transaction_received</code>.</td>
</tr>
<tr>
<td>S_snk_rdyD</td>
<td>Signals the sink is not <code>ready</code>. The event name is <code>signal_snk_ready_deassert</code>.</td>
</tr>
</tbody>
</table>

Block Diagram

Figure 10–3 provides a block diagram of the Avalon-ST Sink BFM. As this figure illustrates, the BFM includes the following five major blocks.

- Avalon-ST Sink API—Provides methods to get Avalon-ST transactions and control the ready signal.
- Transaction Descriptor—Accumulates the fields of an Avalon-ST command.
- Avalon-ST Physical Driver—Asserts and deasserts the ready signal to the system interconnect fabric.
- Physical Bus Monitor—Monitors the physical layer and collects transactions.
- Public Events—Signals the events described in the API.
The Avalon-ST Sink BFM supports all the signals defined for the Avalon-MM sink interface. You can customize the Avalon-ST sink interface using the parameters described in Table 10–2.

For detailed information about the Avalon-ST interface, including timing diagrams, refer to the Avalon Interface Specifications.

Table 10–2. Parameters for the Avalon-ST Sink BFM (Part 1 of 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include the signals to support packets</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes the startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>Use the channel signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes channel pin or pins.</td>
</tr>
<tr>
<td>Use the error signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes error pin or pins.</td>
</tr>
<tr>
<td>Use the ready signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a ready pin.</td>
</tr>
<tr>
<td>Use the valid signal</td>
<td>On</td>
<td>On/Off</td>
<td>When On, the interface includes a valid pin.</td>
</tr>
<tr>
<td>Use the empty signal</td>
<td>Off</td>
<td>On/Off</td>
<td>When On, the interface includes empty pins.</td>
</tr>
</tbody>
</table>
The API provides methods for a testbench which instantiates, controls, and queries state in this BFM component. Test programs must only use these public access methods and events to communicate with this BFM component.

The API includes methods fall into the three categories:

- Methods to initialize the BFM and determine its state
- Methods to get Avalon-ST transactions
- Methods to determine the response of the DUT

The following section presents these methods in alphabetical order.

### get_transaction_channel()

**Prototype:**

```
get_transaction_channel().
```

**Arguments:**

None.

**Returns:**

STChannel_t.

**Description:**

Returns the channel identifier for the most recently popped transaction.
get_transaction_data()

Prototype: get_transaction_data(data).
Arguments: None.
Returns: STData_t.
Description: Returns the data in the most recently popped transaction.

get_transaction_idles()

Prototype: get_transaction_idles().
Arguments: None.
Returns: bit[31:0].
Description: Returns the number of idle cycles in the most recently popped transaction.

get_transaction_eop()

Prototype: get_transaction_eop().
Arguments: None.
Returns: bit.
Description: Returns the transaction end of packet status in the most recently popped transaction.

get_transaction_empty()

Prototype: get_transaction_empty().
Arguments: None.
Returns: STEmpty_t.
Description: Returns the number of empty symbols in the most recently popped transaction.

get_transaction_error()

Prototype: get_transaction_error().
Arguments: None.
Returns: STError_t.
Description: Returns the error in the most recently popped transaction.

get_transaction_queue_size()

Prototype: get_transaction_queue_size().
Arguments: None.
Returns: int.
Description: Returns the length of the queue holding received transactions.
get_transaction_sop()

Prototype: get_transaction_sop().
Arguments: None.
Returns: bit.
Description: Returns the transaction start of packet status in the most recently popped transaction.

get_version()

Prototype: get_version().
Arguments: None.
Returns: String.
Description: This method returns BFM version as a string of three integers separated by periods. For example, version 9.1 SP1 is encoded as "9.1.1".

init()

Prototype: init
Arguments: None.
Returns: void.
Description: Drives the interface to the idle state.

pop_transaction()

Prototype: pop_transaction().
Arguments: None.
Returns: void.
Description: Pops the transaction descriptor from the queue so that the testbench can query it using the get_transaction methods.

set_ready()

Prototype: set_ready().
Arguments: None.
Returns: bit.
Description: Sets the value of the interface ready signal. To assert back pressure, deassert this signal. The parameter USE_READY must be set to 1 to enable the ready signal.

signal_fatal_error

Prototype: signal_fatal_error.
Arguments: None.
Returns: void.
Description: This event signals that a fatal error has occurred. It terminates the simulation.
signal_sink_ready

Prototype: signal_sink_ready.
Arguments: None.
Returns: void.
Description: This event signals that sink_ready is asserted, turning off back pressure.

signal_sink_ready_deassert

Prototype: signal_sink_ready_deassert.
Arguments: None.
Returns: void.
Description: This event signals that sink_ready is deasserted, turning on back pressure.

signal_transaction_received

Prototype: signal_transaction_received.
Arguments: None.
Returns: void.
Description: This event signals that the transaction has been received and queued.
The Avalon-ST Sink BFM with Avalon-ST API Wrapper provides VHDL support for the Avalon-ST Sink BFM. You can use the Avalon-ST Sink BFM with Avalon-ST API Wrapper in HDL simulators that support mixed language simulation. This component is implemented in SystemVerilog and uses an API wrapper to cast the Avalon-ST Sink BFM’s method calls and returns into signals that are carried on the call and return interface ports. The wrapper is necessary because VHDL can only access ports and does not support the method calls used in the Avalon-ST Sink BFM. Figure 11–1 provides a high-level view of this component.

Figure 11–1. Avalon-ST Sink BFM with Avalon-ST Wrapper

In Figure 11–1, the API call interface and Avalon-ST call and return interface operate in separate clock domains with \( \text{av clk} \) synchronizing the FPGA logic and \( \text{api clk} \) synchronizing the Avalon-ST translation interface. Typically, the Avalon-ST interface, which is not part of the actual hardware design, operates at a much higher frequency than the Avalon-ST Sink BFM interface.
For every function call in the BFM, there is a channel identifier which stores the fixed mapping between channel number and the function.  
<install_dir>/ip/altera/sopc_builder_ip/verification/lib/altera_avalon_components_pkg.vhd defines the following function calls:

- ST_SINK_INIT
- ST_SINK_SET_READY
- ST_SINK_POP_TRANS
- ST_SINK_GET_TRANS_IDLES
- ST_SINK_GET_TRANS_DATA
- ST_SINK_GET_TRANS_CHANNEL
- ST_SINK_GET_TRANS_SOP
- ST_SINK_GET_TRANS_EOP
- ST_SINK_GET_TRANS_ERROR
- ST_SINK_GET_TRANS_EMPTY
- ST_SINK_GET_TRANS_QUEUE_SIZE

With the exception of the API wrapper, this component is identical to the Avalon-ST Sink BFM. For more information about this component refer to Chapter 10, Avalon Streaming Sink BFM.
The Avalon-ST Monitor verifies Avalon-ST interfaces using SystemVerilog assertions. In addition, it provides test coverage reports so that you can determine when your test vectors provide sufficient test coverage for your device under test (DUT) functionality. Using the Avalon-ST Monitor has three key advantages:

- It provides independent verification of the Avalon-ST protocol.
- It simplifies the verification task by providing a key component of the verification environment.
- It provides test coverage statistics, so that you know when you have completed sufficient testing.

The Avalon-ST Monitor is implemented in SystemVerilog and uses the SystemVerilog Assertion (SVA) language. SVA is supported by the Synopsis VCS, and Mentor Questa. If you are using ModelSim, the monitor component still compiles and simulates, but the assertion checking is turned off.

Figure 12–1 shows a testbench that uses an Avalon-ST Monitor to test components with Avalon-ST interfaces. As this figure illustrates, the monitor’s Avalon-ST source interface is connected to the DUT’s Avalon-ST sink interface, and an Avalon-ST sink interface is connected to the DUT’s Avalon-ST source interface. The test program communicates with the monitor. It uses the monitor’s assertion checking and coverage groups to assure that all legal parameter values for the DUT’s Avalon-ST interfaces are verified.

Figure 12–1. Testbench Using with an Avalon-ST Monitor
Parameters

The Avalon-ST monitor supports the full range of signals defined for the Avalon-ST source and sink interfaces. You can customize the Avalon-ST source and sink interfaces using the parameters described in Table 12–1.

For detailed information about the Avalon-ST interface, including timing diagrams, refer to the Avalon Interface Specifications.

Table 12–1. Parameters for the Avalon-ST Monitor BFM

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Widths</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbol width</td>
<td>8</td>
<td>On/Off</td>
<td>Data symbol width in bits. The symbol width should be 8 for byte-oriented interfaces.</td>
</tr>
<tr>
<td>Number of symbols</td>
<td>4</td>
<td>On/Off</td>
<td>Numbers of symbols per word.</td>
</tr>
<tr>
<td>Width of the channel signal</td>
<td>1</td>
<td>On/Off</td>
<td>Specifies the width of the channel signal n bits.</td>
</tr>
<tr>
<td>Width of the error signal</td>
<td>1</td>
<td>On/Off</td>
<td>Specifies the width of the error signal n bits.</td>
</tr>
<tr>
<td><strong>Port Enables</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Include the signals to support packets</td>
<td>On/Off</td>
<td>On/Off</td>
<td>When On, the interface includes the startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>Use the channel signal</td>
<td>On/Off</td>
<td>On/Off</td>
<td>When On, the interface includes a channel pin.</td>
</tr>
<tr>
<td>Use the error signal</td>
<td>On/Off</td>
<td>On/Off</td>
<td>When On, the interface includes error pins.</td>
</tr>
<tr>
<td>Use the ready signal</td>
<td>On/Off</td>
<td>On/Off</td>
<td>When On, the interface includes ready pins.</td>
</tr>
<tr>
<td>Use the valid signal</td>
<td>On/Off</td>
<td>On/Off</td>
<td>When On, the interface includes valid pins.</td>
</tr>
<tr>
<td>Use the empty signal</td>
<td>On/Off</td>
<td>On/Off</td>
<td>When On, the interface includes an empty pin.</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready latency</td>
<td>0</td>
<td>On/Off</td>
<td>Specifies the readyLatency parameter for data interfaces that support backpressure. Refer to the Avalon Interface Specifications for more information.</td>
</tr>
<tr>
<td>Max Channel Number</td>
<td>1</td>
<td>On/Off</td>
<td>Specifies when a timeout will occur if a burst write transfer has not completed.</td>
</tr>
</tbody>
</table>

Application Program Interface (API)

The Avalon-ST Monitor’s API methods to enable and disable protocol assertions which are used to ensure protocol compliance. For example, the enable_a_no_data_outside_packet method enables the assertion that verifies that no data is transmitted between the assertion of the endofpacket and the next startofpacket signals. If a violation is found, an error message is displayed on the console running the simulation. Error flags also are displayed in the waveform viewer.
Coverage groups ensure that the verification suite test all expected functionality of the interface. Methods such as the `cover_b2b_packet_different_channel` method allow each individual coverage point to be enabled or disabled. When coverage points are disabled, they do not show up as missing coverage in the coverage report.

By default, each assertion and coverage point that is appropriate to the parameterization is enabled. For example, if the interface does not use packets, the coverage groups that test packet transfers are automatically disabled. Occasionally some assertion checking may be turned off to avoid triggering assertions when injecting protocol errors to test error handling capability.

The names of all methods that implement assertions begin with `enable_a`. The names of all methods that implement coverage functionality begin with `enable_c`.

By default, if your testbench includes the Avalon-ST monitor, the checking function is enabled. You can disable checking with the `DISABLE_ALTERA_AVALON_SIM_SVA` macro.

To generate the coverage report when using the Synopsys VCS simulator, you can execute the following command:

```
    urg -dir simv.vdb
```

### enable_a_empty_legal()

**Prototype:**

```
enable_a_empty_legal().
```

**Arguments:**

Boolean.

**Returns:**

Void.

**Description:**

This method enables an assertion that ensures `empty` is 0 except when `endofpacket` is asserted and that `empty` is always less than the number of symbols in a packet.

### enable_a_less_than_max_channel()

**Prototype:**

```
enable_a_less_than_max_channel().
```

**Arguments:**

Boolean.

**Returns:**

Void.

**Description:**

This method enables an assertion that ensures that the value of the `channel` signal is less than the maximum number of channels.

### enable_a_no_data_outside_packet()

**Prototype:**

```
enable_a_no_data_outside_packet().
```

**Arguments:**

Boolean.

**Returns:**

Void.

**Description:**

This method enables an assertion that ensures `valid` data is not transferred outside of a packet when the interface uses packet transmission.
enable_a_non_missing_endofpacket()

Prototype: enable_a_non_missing_endofpacket().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that the startofpacket signal is asserted between each two assertions of an endofpacket signal.

enable_a_non_missing_startofpacket()

Prototype: enable_a_non_missing_startofpacket().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures that each assertion of the startofpacket signal is followed by the assertion of an endofpacket signal.

enable_a_valid_legal()

Prototype: enable_a_valid_legal().
Arguments: Boolean.
Returns: Void.
Description: This method enables an assertion that ensures valid is deasserted readyLatency cycles after ready is deasserted if the readyLatency is greater than 0.

enable_c_b2b_data_different_channel()

Prototype: enable_c_b2b_data_different_channel().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures back-to-back valid signals for different channels. It is disabled when channels are not supported.

enable_c_b2b_data_same_channel()

Prototype: enable_c_b2b_data_same_channel().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage for back-to-back valid signals for the same channel. It is disabled when channels are not supported.
enable_c_b2b_packet_different_channel()

Prototype: enable_c_b2b_packet_different_channel().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage for back-to-back packet transmission for different channels. It is disabled when packet transmission or channels are not supported.

enable_c_b2b_packet_same_channel()

Prototype: enable_c_b2b_packet_same_channel().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage for back-to-back packet transmission for the same channel. It is disabled when packet transmission or channels are not supported.

enable_c_channel_change_in_packet()

Prototype: enable_c_channel_change_in_packet().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage of a change of channels within the packet transaction. It is disabled when either the channel signal or packet transmission is not supported.

enable_c_empty()

Prototype: enable_c_empty().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage of a empty signal. It is disabled when packet transmission is not supported.

enable_c_error()

Prototype: enable_c_error().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage of all bits of the error signal. It is disabled when the error signal is not supported.
enable_c_non_valid_ready()

Prototype: enable_c_non_valid_ready().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage for the assertion of valid signal with different values for readyLatency. Refer to the Avalon Interface Specifications for more information.

enable_c_non_valid_non_ready()

Prototype: enable_c_non_valid_non_ready().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage for the deassertion of both ready and valid. It is disabled when the ready signal is not supported.

enable_c_packet()

Prototype: enable_c_packet().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage packet transmission for different values of the channel signal. It is disabled when packet transmission is not supported.

enable_c_packet_no_idles_no_back_pressure()

Prototype: enable_c_packet_no_idles_no_back_pressure().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage of packet transaction without back pressure and idle cycles. It is disabled when packet transmission is not supported.

enable_c_packet_with_back_pressure()

Prototype: enable_c_packet_with_back_pressure().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage of packet transaction with backpressure. It is disabled when either the ready signal or packet transmission is not supported.
enable_c_packet_with_idles()

Prototype: enable_c_packet_with_idles().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage of packet transaction with idle cycles. It is disabled when packet transmission is not supported.

enable_c_transfer()

Prototype: enable_c_transfer().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage of a valid signal is asserted correctly for different channels. It is disabled when the ready or valid signals are not supported.

enable_c_valid_non_ready()

Prototype: enable_c_valid_non_ready().
Arguments: Boolean.
Returns: Void.
Description: This method enables a coverage point that ensures test coverage for valid signal when ready is deasserted. It is disabled when the readyLatency is greater than 0.
Additional Information

Revision History

The following table shows the revision history for this user guide.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes Made</th>
</tr>
</thead>
</table>

How to Contact Altera

For the most up-to-date information about Altera® products, see the following table.

<table>
<thead>
<tr>
<th>Contact (Note 1)</th>
<th>Contact Method</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical support</td>
<td>Website</td>
<td><a href="http://www.altera.com/support">www.altera.com/support</a></td>
</tr>
<tr>
<td>Technical training</td>
<td>Website</td>
<td><a href="http://www.altera.com/training">www.altera.com/training</a></td>
</tr>
<tr>
<td>Email</td>
<td></td>
<td><a href="mailto:custrain@altera.com">custrain@altera.com</a></td>
</tr>
<tr>
<td>Altera literature services</td>
<td>Email</td>
<td><a href="mailto:literature@altera.com">literature@altera.com</a></td>
</tr>
<tr>
<td>Non-technical support (General)</td>
<td>Email</td>
<td><a href="mailto:nacomp@altera.com">nacomp@altera.com</a></td>
</tr>
<tr>
<td>(Software Licensing)</td>
<td>Email</td>
<td><a href="mailto:authorization@altera.com">authorization@altera.com</a></td>
</tr>
</tbody>
</table>

Note:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

<table>
<thead>
<tr>
<th>Visual Cue</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold Type with Initial Capital Letters</strong></td>
<td>Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: <em>Save As</em> dialog box.</td>
</tr>
<tr>
<td><strong>bold type</strong></td>
<td>External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: ( f_{\text{MAX}} ), ( \text{qdesigns} ) directory, ( d: ) drive, <em>chiptrip.gdf</em> file.</td>
</tr>
<tr>
<td><strong>Italic Type with Initial Capital Letters</strong></td>
<td>Document titles are shown in italic type with initial capital letters. Example: <em>AN 75: High-Speed Board Design</em>.</td>
</tr>
<tr>
<td><strong>Italic type</strong></td>
<td>Internal timing parameters and variables are shown in italic type. Examples: ( t_{\text{PAK}}, n + 1 ). Variable names are enclosed in angle brackets (&lt; &gt;) and shown in italic type. Example: <em>&lt;file name&gt;, &lt;project name&gt;.pof</em> file.</td>
</tr>
<tr>
<td>Initial Capital Letters</td>
<td>Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.</td>
</tr>
<tr>
<td>Visual Cue</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>“Subheading Title”</td>
<td>References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”</td>
</tr>
<tr>
<td>Courier type</td>
<td>Signal and port names are shown in lowercase Courier type. Examples: data1, tdi, input. Active-low signals are denoted by suffix n, e.g., resetn. Anything that must be typed exactly as it appears is shown in Courier type. For example: c:\qdesigns\tutorial\chiptrip.gdf. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword SUBDESIGN), as well as logic function names (e.g., TRI) are shown in Courier.</td>
</tr>
<tr>
<td>1., 2., 3., and a., b., c., etc.</td>
<td>Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.</td>
</tr>
<tr>
<td>■ ■</td>
<td>Bullets are used in a list of items when the sequence of the items is not important.</td>
</tr>
<tr>
<td>☑</td>
<td>The checkmark indicates a procedure that consists of one step only.</td>
</tr>
<tr>
<td>I.</td>
<td>The hand points to information that requires special attention.</td>
</tr>
<tr>
<td>CAUTION</td>
<td>A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.</td>
</tr>
<tr>
<td>WARNING</td>
<td>A warning calls attention to a condition or possible situation that can cause injury to the user.</td>
</tr>
<tr>
<td></td>
<td>The angled arrow indicates you should press the Enter key.</td>
</tr>
<tr>
<td></td>
<td>The feet direct you to more information on a particular topic.</td>
</tr>
</tbody>
</table>