SoC Designer
Version 9.1.0

AHBv2 Protocol Bundle
User Guide

Copyright © 2017 ARM Limited. All Rights Reserved
ARM DUI 1080A
SoC Designer
AHBv2 Protocol Bundle User Guide

Copyright © 2017 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>February 2017</td>
<td>A</td>
<td>Non-Confidential</td>
<td>Restamp release.</td>
</tr>
</tbody>
</table>

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM Limited (ARM). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any patents.

THIS DOCUMENT IS PROVIDED AS IS. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version shall prevail.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word partner in reference to ARM’s customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the ARM trademark usage guidelines http://www.arm.com/about/trademarks/guidelines/index.php.

Copyright © ARM Limited or its affiliates. All rights reserved.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

In this document, where the term ARM is used to refer to the company it means ARM or any of its subsidiaries as appropriate.
Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com
## Table of Contents

1 **Introduction** ............................................................................................ 6

2 **Requirements** .......................................................................................... 6

3 **Package Contents**.................................................................................... 6
   3.1 **AHBv2 Models and Examples**.............................................................. 6
   3.2 **AHBv2 Probes** ........................................................................................ 6
   3.3 **AHBv2 Transactors** .............................................................................. 6
   3.4 **AHBv2 Ports** ....................................................................................... 6
   3.5 **AHBv2 Component Wizard Templates**................................................. 6

4 **Models**...................................................................................................... 7
   4.1 **AHBv2_Master** .................................................................................... 8
   4.2 **AHBv2_Slave** .................................................................................... 9
   4.3 **AHBv2_Mem and AHBv2_LiteMem**..................................................... 10
   4.4 **AHBv2_Slub** .................................................................................... 12
      4.4.1 **Stub Macros** .................................................................................. 12
      4.4.2 **Simulation Features** ..................................................................... 13
         4.4.2.1 **Register** .................................................................................. 13
         4.4.2.2 **Memory** .................................................................................. 13
         4.4.2.3 **Disassembly** .......................................................................... 13
   4.5 **MxAHBv2 and MxAHBv2_Lite**............................................................ 14
      4.5.1 **Arbitration** ................................................................................... 14
   4.6 **AHBv2ToAPB and AHBv2LiteToAPB** ................................................. 16
   4.7 **AHBv2ToAHBv2LiteSS and AHBv2LiteToAHBv2SS**.......................... 17
   4.8 **AHBv2ToAHBv2LiteMS** .................................................................... 18
   4.9 **AHBv2ToMx and AHBv2LiteToMx**...................................................... 19
4.10 MxToAHBv2 and MxToAHBv2Lite ................................................................. 20
4.11 AHBv2Mux and AHBv2Mux_Lite ................................................................. 21

5 Probes ........................................................................................................... 22
5.1 Tracer .......................................................................................................... 22
5.2 Breakpoint .................................................................................................. 23
5.3 Profiling ...................................................................................................... 23

6 Component Wizard .................................................................................... 24
6.1 Generating AHBv2 Ports ........................................................................... 24

7 Transactors .................................................................................................. 25
7.1 AHBv2 Transactors and Cycle Model Studio ............................................. 26

8 AHBv2 Port Interfaces ................................................................................... 26
8.1 AMBA AHB Interfaces AHBv2 Transaction Ports ...................................... 26
  8.1.1 AHB_Master_Port .................................................................................. 27
    8.1.1.1 AHB_Master_PortBase ................................................................. 28
    8.1.1.2 AHB_Master_Port Template Specialization ..................................... 29
  8.1.2 AHB_Slave_Port .................................................................................. 32
    8.1.2.1 AHB_Slave_PortBase ................................................................. 32
    8.1.2.2 AHB_Slave_Port Template Specialization ..................................... 34
  8.1.3 AHBPortIF .......................................................................................... 37
  8.1.4 Transaction Phases ............................................................................... 37
  8.1.5 Examples ............................................................................................. 37
8.2 AHB Extensions .......................................................................................... 38
  8.2.1 Cortex-M3 Sideband Signals ................................................................. 38
1 Introduction

This is the user guide for SoC Designer AHBv2 Protocol Bundle. This protocol bundle contains SoC Designer components, probes, and transactors for the ARM AMBA AHB transaction protocol.

2 Requirements

AHBv2 protocol bundle requires the following:

- SoC Designer v9.0.0
- Compilation tools as set forth in the SoC Designer Installation Guide (ARM DUI 0953).

3 Package Contents

AHBv2 protocol bundle contains the components described in this section.

3.1 AHBv2 Models and Examples

Generic components such as configurable bus and memory are included in this bundle. Also provided is example source code to help you develop custom AHB components.

3.2 AHBv2 Probes

Probes provide visibility into transactions between two components. AHBv2-specific probes are included in this protocol bundle.

3.3 AHBv2 Transactors

Transactors bridge the signal and transaction level communication. AHBv2 transactors are included in this protocol bundle, and can be used by Cycle Model Studio to generate SoC Designer components with AHBv2 transaction ports.

3.4 AHBv2 Ports

AHBv2 transaction port definition header files and libraries are included in this package. These are required during runtime of any components with AHBv2 ports and also when creating components with AHBv2 ports.

3.5 AHBv2 Component Wizard Templates

The AHBv2 protocol bundle includes the template files needed by the SoC Designer Component Wizard for generation of components with AHBv2 ports.
## 4 Models

The following table lists the AHBv2 components included in this bundle.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHBv2_Master</td>
<td>This is an example AHBv2 master component. This component is available in source code format.</td>
</tr>
<tr>
<td>AHBv2_Slave</td>
<td>This is an example AHBv2 slave component. This component is available in source code format.</td>
</tr>
<tr>
<td>AHBv2_Mem</td>
<td>This is a generic AHB memory model with an AHBv2 transaction slave port.</td>
</tr>
<tr>
<td>AHBv2_LiteMem</td>
<td>An AHB-Lite version of AHBv2_Mem.</td>
</tr>
<tr>
<td>AHBv2_Stub</td>
<td>This is a scriptable (reads in *.mxscr) AHB master component.</td>
</tr>
<tr>
<td>MxAHBv2</td>
<td>A generic AHB bus component which can be configured for up to 16 masters and 16 slaves.</td>
</tr>
<tr>
<td>MxAHBv2_Lite</td>
<td>An AHB-Lite version of MxAHBv2. Supports up to 16 slaves.</td>
</tr>
<tr>
<td>AHBv2ToAPB</td>
<td>A bridge component which translates AHB to APB transactions.</td>
</tr>
<tr>
<td>AHBv2LiteToAPB</td>
<td>A bridge component which translates AHB-Lite slave to APB transactions.</td>
</tr>
<tr>
<td>AHBv2ToAHBv2LiteSS</td>
<td>A bridge component that allows AHB-Lite slaves to be hooked up to an AHB bus.</td>
</tr>
<tr>
<td>AHBv2LiteToAHBv2SS</td>
<td>A bridge component that allows AHB slaves to be hooked up to an AHB-Lite bus.</td>
</tr>
<tr>
<td>AHBv2ToAHBv2LiteMS</td>
<td>A bridge component that allows an AHB master to be hooked up to an AHB-Lite system.</td>
</tr>
<tr>
<td>AHBv2ToMx</td>
<td>AHBv2 to MX protocol conversion bridge.</td>
</tr>
<tr>
<td>AHBv2LiteToMx</td>
<td>AHBv2-Lite to MX protocol conversion bridge.</td>
</tr>
<tr>
<td>MxToAHBv2</td>
<td>MX to AHBv2 protocol conversion bridge.</td>
</tr>
<tr>
<td>MxToAHBv2Lite</td>
<td>MX to AHBv2-Lite protocol conversion bridge.</td>
</tr>
<tr>
<td>AHBv2Mux</td>
<td>AHB slave multiplexor for multi-layer AHB designs.</td>
</tr>
</tbody>
</table>

Table 4-1  AHBv2 Components
4.1 AHBv2_Master

AHBv2_Master is an example AHBv2 master component which is available in source code format. This component initiates a configurable number of AHB NONSEQ transfers over incrementing addresses.

The source code is located in $MAXSIM_PROTOCOLS/AHBv2/src/AHBv2_Master. There is also an example system which uses AHBv2_Master, which is located in $MAXSIM_PROTOCOLS/AHBv2/examples/MasterSlave_2x2.

The table below lists the component parameters.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Width</td>
<td>Data bus width in bits. Supported values are 32, 64, and 128.</td>
</tr>
<tr>
<td>Enable Debug Messages</td>
<td>Boolean flag to enable/disable debug messages.</td>
</tr>
<tr>
<td>Enable Verbosity</td>
<td>Boolean flag to enable/disable messages related to transactions generated by this component.</td>
</tr>
<tr>
<td>Num Transactions</td>
<td>Total number of transactions to generate.</td>
</tr>
<tr>
<td>Start Address</td>
<td>The starting address of transactions. The address will be incremented by the Data Width/8 for each transaction.</td>
</tr>
</tbody>
</table>

Table 4-2 AHBv2_Master parameters
4.2 AHBv2_Slave

AHBv2_Slave is an example AHBv2 slave component which is available in source code format. This component behaves as a simple memory device.

The source code is located in $MAXSIM_PROTOCOLS/AHBv2/src/AHBv2_Slave. There is also an example system which uses AHBv2_Slave, which is located in $MAXSIM_PROTOCOLS/AHBv2/examples/MasterSlave_2x2.

![AHBv2_Slave Diagram]

Figure 4-2 AHBv2_Slave

The table below lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Debug Messages</td>
<td>Boolean flag to enable/disable debug messages.</td>
</tr>
<tr>
<td>Enable Verbosity</td>
<td>Boolean flag to enable/disable messages related to transactions received by this component.</td>
</tr>
<tr>
<td>region name</td>
<td>Name of the memory region occupied by this AHB slave.</td>
</tr>
<tr>
<td>region size</td>
<td>Size of the region occupied by this AHB slave.</td>
</tr>
<tr>
<td>region start</td>
<td>The base address of the memory region occupied by this AHB slave.</td>
</tr>
</tbody>
</table>

Table 4-3 AHBv2_Slave parameters
### 4.3 AHBv2_Mem and AHBv2_LiteMem

AHBv2_Mem is a generic AHB memory model with an AHB slave interface. AHBv2_LiteMem is the same as AHBv2_Mem, except that it includes an AHB-Lite slave interface.

![ahb2_mem](ahb2_mem.png)

**Figure 4-3 AHBv2_Mem**

The table below lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 32bit aligned         | \(true\) = accesses are aligned to 32bit addresses. The lower 2 bits of the byte addresses are ignored, and the parameter byte addressable must be \(true\).  
\(false\) = accesses can use byte addresses without any particular alignment. |
| AHB extra warnings    | Additional access checks specific to AHB. This should be enabled for debugging purposes only as it decreases simulation speed. |
| ahb_name\(X\)         | Name of the memory region \(X\)                                                              |
| ahb_size\(X\)         | Size of the memory region \(X\)                                                              |
| ahb_start\(X\)        | The base address of the memory region \(X\)                                                  |
| big endian            | \(true\) = byte order is big endian.  
\(false\) = byte order is little endian                                                     |
| byte addressable      | \(true\) = addresses refer to byte addresses.  
\(false\) = addresses refer to halfword addresses                                            |
| casLatency(r)         | Number of latency cycles when accessing a new column during reads                           |
| casLatency(w)         | Number of latency cycles when accessing a new column during writes                          |
| colBits               | Number of bits in an address describing the column (counted from LSB). Note: pageBits + colBits + rowBits must be 32 |
| data width            | Data bus width. Allowed values are 32, 64, and 128.                                           |
| enable debug messages | Enable extra debug messages                                                                 |

Non-Confidential
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init value</td>
<td>Value assigned to memory during initialization</td>
</tr>
<tr>
<td>pageBits</td>
<td>Number of bits in an address describing the page (counted from MSB). Note: pageBits + colBits + rowBits must be 32.</td>
</tr>
<tr>
<td>pageLatency(r)</td>
<td>Number of latency cycles when accessing a new page during reads</td>
</tr>
<tr>
<td>pageLatency(w)</td>
<td>Number of latency cycles when accessing a new page during writes</td>
</tr>
<tr>
<td>rasLatency(r)</td>
<td>Number of latency cycles when accessing a new row during reads.</td>
</tr>
<tr>
<td>rasLatency(w)</td>
<td>Number of latency cycles when accessing a new row during writes.</td>
</tr>
<tr>
<td>read only</td>
<td>( \text{true} = ) the memory is read only except via the debug interfaces</td>
</tr>
<tr>
<td>refresh ratio</td>
<td>Number of cycles after which a refresh cycle is modeled</td>
</tr>
<tr>
<td>rowBits</td>
<td>Number of bits in an address describing the row (counted from MSB-pageBits). Note: pageBits + colBits + rowBits must be 32.</td>
</tr>
<tr>
<td>use input file</td>
<td>When this parameter is set to ( \text{true} ), the model can be initialized with a binary file containing a memory image. The binary file consists of values only. The addresses are assumed to start at the base address and are incremented with each value. ( \text{true} = ) prompt for input file on hard reset ( \text{false} = ) no input file</td>
</tr>
</tbody>
</table>

Table 4-4  AHBv2_Mem parameters
## 4.4 AHBv2_Stub

AHBv2_Stub is an AHBv2 master component which can be controlled with a SoC Designer .mxscr script. AHB transactions are generated on the ahb2_m transaction master port. The ahb2_s port can be used to drive AHB transactions.

AHBv2_Stub has an internal memory that you can use as a place holder for an AHB slave. Like other stub components, AHBv2_Stub has a number of signal masters and slaves that can be controlled from an .mxscr script.

An example system is located in $MAXSIM_PROTOCOLS/AHBv2/examples/AHBv2_Stub.

### Table 4-5 AHBv2_Stub parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ahb_startX</td>
<td>Start address of the AHB memory regions.</td>
</tr>
<tr>
<td>CPP include path</td>
<td>Additional include path for header files to be used by script preprocessor.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data bus width. Supported values are 32, 64, and 128.</td>
</tr>
<tr>
<td>Enable Debug Messages</td>
<td>Enable extra debug messages.</td>
</tr>
<tr>
<td>Memory Init Byte Value</td>
<td>Initialization value for the internal memory (when acting as AHB slave)</td>
</tr>
</tbody>
</table>

### 4.4.1 Stub Macros

Convenience macros for AHB traffic generation are available in AHBv2_Stub_Macros.h located in $MAXSIM_PROTOCOLS/AHBv2/include. For backward compatibility to AHBv1, all macros defined in the original AHBv1 stub macros have been redefined, which means that you can reuse the old AHBv1 stub scripts by replacing the #include line to include AHBv2_Stub_Macros.h (instead of AHB_Stub_Macros.h).
4.4.2 Simulation Features

4.4.2.1 Register
The AHBv2_Stub register window shows information about the currently-loaded .mxscr script as well as stub internal data which may be useful in debugging user-defined stub macros.

4.4.2.2 Memory
The AHBv2_Stub memory window has three address spaces:
- MxStub - Internal use only (ignore)
- Memory - AHBv2_Stub internal memory
- AHB-Master - External memory seen by the ahb2_m port

4.4.2.3 Disassembly
AHBv2_Stub supports the disassembly view which shows the .mxscr script file and the commands being executed on the stub. As with other disassembly windows, simulation control buttons as well as instruction (or a script command) cycle counts are available from this window.
4.5 MxAHBv2 and MxAHBv2_Lite

MxAHBv2 is an AHB bus component which can be configured up to 16 masters and 16 slaves. MxAHBv2_Lite is an AHB-Lite version of the bus which can support up to 16 slaves.

Unused ports can be disabled and hidden in SoC Designer Canvas. To do this:

1. Make the port connections for the ports in use.
2. Right-click on the component in SoC Designer Canvas.
3. From the context menu, select **Disable All Unconnected Ports**.
4. Select **Hide All Disabled Ports**.

4.5.1 Arbitration

Arbitration policy can be configured to either a round-robin or a fixed-priority scheme. Fixed priority uses the master 0 port as the highest priority master (master 15 is the lowest priority).

For arbitration to work correctly, slave ports must be used consecutively starting with the ahb_s00 port. For example, connecting a master to ahb_s01 and leaving ahb_s00 unconnected is not allowed (an error message is issued at the beginning of simulation).

![Figure 4-5 MxAHBv2](image)

<table>
<thead>
<tr>
<th>mxaahbv2[0] (MxAHBv2)</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>ahb_s00</td>
<td>ahb_m00</td>
</tr>
<tr>
<td>ahb_s01</td>
<td>ahb_m01</td>
</tr>
<tr>
<td>ahb_s02</td>
<td>ahb_m02</td>
</tr>
<tr>
<td>ahb_s03</td>
<td>ahb_m03</td>
</tr>
<tr>
<td>ahb_s04</td>
<td>ahb_m04</td>
</tr>
<tr>
<td>ahb_s05</td>
<td>ahb_m05</td>
</tr>
<tr>
<td>ahb_s06</td>
<td>ahb_m06</td>
</tr>
<tr>
<td>ahb_s07</td>
<td>ahb_m07</td>
</tr>
<tr>
<td>ahb_s08</td>
<td>ahb_m08</td>
</tr>
<tr>
<td>ahb_s09</td>
<td>ahb_m09</td>
</tr>
<tr>
<td>ahb_s10</td>
<td>ahb_m10</td>
</tr>
<tr>
<td>ahb_s11</td>
<td>ahb_m11</td>
</tr>
<tr>
<td>ahb_s12</td>
<td>ahb_m12</td>
</tr>
<tr>
<td>ahb_s13</td>
<td>ahb_m13</td>
</tr>
<tr>
<td>ahb_s14</td>
<td>ahb_m14</td>
</tr>
<tr>
<td>ahb_s15</td>
<td>ahb_m15</td>
</tr>
<tr>
<td>clk-in</td>
<td></td>
</tr>
</tbody>
</table>
The table below lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Arbitration Policy         | Round-Robin or Priority. Default is Round-Robin. Priority-based arbitration sets master 0 (master connected on ahb_s00 port) as the highest priority master, and master 15 (master connected on ahb_s15) as the lowest priority master.  
  *Note: Available for MxAHBv2 only.* |
| Data Width                  | Data bus width. Supported values are 32, 64, and 128.                                                                                       |
| Enable Debug Messages       | Enable extra debug messages.                                                                                                                |
| Use MME                     | Use Memory Map Editor for memory map configuration.                                                                                         |
|                             | *Note: Memory regions reported by the slaves are ignored when MME is in use.*                                                               |
| HTRANS IDLE when HSEL==0    | When set to true, HTRANS will be forced to IDLE on the slave side when the slave is not selected (i.e., HSEL==0).                           |

Table 4-6 MxAHBv2 parameters
4.6 AHBv2ToAPB and AHBv2LiteToAPB

These components map AHB or AHB-Lite slave interface to APB transactions.

Connect the ahbv2 port to the master port of MxAHBv2 (or MxAHBv2_Lite for AHB-Lite). The apb port is a bus master port, which means it can be hooked up to multiple APB slaves.

Both variants of the bridge support AMBA3 extensions of APB (PREADY and PSLVERR responses are supported).

Table 4-7 lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Debug Messages</td>
<td>Boolean flag to enable/disable debug messages.</td>
</tr>
<tr>
<td>APB region base</td>
<td>Base address of the peripheral region.</td>
</tr>
<tr>
<td>APB region size</td>
<td>Size of the peripheral region.</td>
</tr>
<tr>
<td>use MME</td>
<td>Use Memory Map Editor for setting up the memory map.</td>
</tr>
</tbody>
</table>

Table 4-7 AHBv2(Lite)ToAPB parameters
4.7 AHBv2ToAHBv2LiteSS and AHBv2LiteToAHBv2SS

These components bridge between full AHB and AHB-Lite slave interfaces. Use AHBv2ToAHBv2LiteSS to hook up an AHB-Lite slave to an AHB bus. Conversely, use AHBv2LiteToAHBv2SS to hook up a regular AHB slave to an AHB-Lite bus. AHB signals that only exist in the full AHB interface (HRESP[1]) are assumed to be tied to zero when going through these bridges.

![Figure 4-8 AHBv2LiteToAHBv2SS](image)

![Figure 4-9 AHBv2ToAHBv2LiteSS](image)

Table 4-8 lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Debug Messages</td>
<td>Boolean flag to enable/disable debug messages.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data bus width. Supported values are 32, 64, and 128.</td>
</tr>
<tr>
<td>AHB region base</td>
<td>Base address of the AHB slave.</td>
</tr>
<tr>
<td>AHB region size</td>
<td>Size of the AHB slave address region.</td>
</tr>
</tbody>
</table>

Table 4-8 AHBv2ToAHBv2LiteSS and AHBv2LiteToAHBv2SS parameters

[^1]: ARM DUI 1080A Copyright© 2017 ARM Limited. All Rights Reserved Non-Confidential
4.8 AHBv2ToAHBv2LiteMS

This bridge enables a full AHB master to connect to an AHB-Lite bus system. An example system is shown in Figure 4-13.

![Figure 4-10 AHBv2ToAHBv2LiteMS](image)

Table 4-9 lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Debug Messages</td>
<td>Boolean flag to enable/disable debug messages.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data bus width. Supported values are 32, 64, and 128.</td>
</tr>
</tbody>
</table>

Table 4-9 AHBv2ToAHBv2LiteMS parameters
4.9 AHBv2ToMx and AHBv2LiteToMx

These are protocol conversion bridges that enable AHBv2 and AHBv2-Lite masters to connect to MX components. The AHBv2 ports on these components implement the AHB master interface. An interconnect error is issued if these AHBv2 ports are connected to AHBv2 transaction master ports that implement the AHB slave interface (transaction master ports on MxAHBv2).

![Figure 4-11 AHBv2ToMx](image)

![Figure 4-12 AHBv2LiteToMx](image)

The table below lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Debug Messages</td>
<td>Boolean flag to enable/disable debug messages.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data bus width. Supported values are 32 and 64. Note the data width must match between AHBv2 and the MX sides.</td>
</tr>
</tbody>
</table>

Table 4-10 AHBv2ToMx and AHBv2LiteToMx component parameters
4.10 MxToAHBv2 and MxToAHBv2Lite

These are protocol conversion bridges that enable AHBv2 and AHBv2-Lite slaves to connect to MX components. The AHBv2 ports on these components implement the AHB slave interface. An interconnect error is issued if these AHBv2 ports are connected to AHBv2 transaction slave ports that implement the AHB master interface (transaction slave ports on MxAHBv2).

Table 4-11 lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Debug Messages</td>
<td>Boolean flag to enable/disable debug messages.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data bus width. Supported values are 32 and 64. Note the data width must match between AHBv2 and the MX sides. An error is issued for MX transactions that request data transfer greater than allowed by the configured data width.</td>
</tr>
<tr>
<td>hmaster</td>
<td>Value of HMASTER signal to drive onto the AHBv2 slave. Default is 0.</td>
</tr>
<tr>
<td>hprot</td>
<td>Value of HPROT signal to drive onto the AHBv2 slave. Default is 0.</td>
</tr>
</tbody>
</table>

Table 4-11 MxToAHBv2 and MxToAHBv2Lite component parameters
4.11 AHBv2Mux and AHBv2Mux_Lite

These are AHB slave-side bridges that enable access to a single AHB slave from multiple AHB layers. The bridge arbitrates between the AHB master layers when there are concurrent accesses. The arbitration is priority based, where the layer connected on the lower-numbered slave port on the bridge has higher priority; in other words, the layer connected on port 0 has the highest priority. Arbitration takes place between whole transfers: burst transfers and back-to-back transactions are not interrupted even when there are higher-priority master requests for access.

Up to 16 layers are supported. Lower-numbered ports must be used first. Unused, higher-numbered ports can be disabled. An example multi-layer system using AHBv2Mux is shown in Figure 4-18.

Table 4-12 lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ahb_sizeX</td>
<td>Size of slave region X. Up to 6 regions are supported.</td>
</tr>
<tr>
<td>ahb_startX</td>
<td>Base address of slave region X.</td>
</tr>
<tr>
<td>Data Width</td>
<td>AHB data width. Supported values are 32, 64 and 128.</td>
</tr>
</tbody>
</table>

Table 4-12 AHBv2Mux and AHBv2Mux_Lite component parameters
5 Probes

The simulation probes listed in Table 5-1 are included in the AHBv2 Protocol Bundle.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHBv2 Tracer</td>
<td>Enables tracing of AHB signals on an AHBv2 connection. Traced signals can be viewed in the SoC Designer simulator waveform window.</td>
</tr>
<tr>
<td>AHBv2 BreakPoint</td>
<td>Transaction breakpoint on an AHBv2 connection.</td>
</tr>
<tr>
<td>AHBv2 Profiler</td>
<td>Profiles AHBv2 transactions. Profiled data can be viewed in the SoC Designer simulator profiler window.</td>
</tr>
</tbody>
</table>

Table 5-1  AHBv2 probes

5.1 Tracer

This probe allows tracing of AHB signals. Traced signals can be viewed in the SoC Designer waveform window. To add a tracer probe, right-click on an AHBv2 connection and select Enable/Disable Tracing. This brings up the Tracer Properties dialog (Figure 5-1).

![Tracer properties](image.jpg)

By default, all signals are traced. Disable and enable tracing using the checkboxes located on the left side of the signal.
5.2 Breakpoint

To insert a breakpoint probe, either double-click on the connection or right-click on the connection and select Insert/Remove Breakpoint. By default, the breakpoint is activated and breaks on any active AHB transaction across the connection. To configure breakpoint conditions, bring up the breakpoint property dialog by right-clicking on the connection and selecting Edit Breakpoint Properties. The Breakpoint Condition dialog appears (Figure 5-2).

![Figure 5-2 Breakpoint properties](image)

5.3 Profiling

The Profiling Probe enables latency profiling over an AHBv2 connection. To enable this probe:

1. Right-click on a connection and select Profiler, then Enable. The Display option launches the Operation vs. Cycles profiling window.

Refer to the SoC Designer User Guide (ARM DUI 0956) for more information.

To view other available profiling streams, open the Profiling Manager and locate the connection to which the profiling probe was attached. There are three separate streams available for profiling an AHBv2 connection:

- Events: plots the transactions over cycles
- Latency: latency for each operation type
- Address: plots the accessed address location
6 Component Wizard

The SoC Designer Component Wizard allows generation of AHBv2 master and slave ports.  

Note: Refer to the SoC Designer User Guide (ARM DUI 0956) for general information regarding the Component Wizard.

6.1 Generating AHBv2 Ports

To generate a model with AHBv2 ports, launch the component wizard from SoC Designer Canvas and proceed to the port definition step. Click New to create a new port, and select the desired AHBv2 port type from the port type drop-down list. See the SoC Designer User Guide (ARM DUI 0956) for more information.

To select the correct port type, refer to Figure 8-2 AHBv2 port types and Figure 8-3 AHBv2 AHB-Lite port types in this document.

The model generation process generates a .cpp and a .h file for each AHBv2 port that was selected. The port class inherits from one of the specialized template AHBv2 port classes.
7 Transactors

Table 7-1 lists the SoC Designer transactors included in the AHBv2 Protocol Bundle.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHB_Slave_FT2S</td>
<td>A transaction-to-signal transactor for a transaction slave port on the slave side of a full AHB bus</td>
</tr>
<tr>
<td>AHB_Slave_FS2T</td>
<td>A signal-to-transaction transactor for a transaction master port on the slave side of a full AHB bus</td>
</tr>
<tr>
<td>AHB_Master_FT2S</td>
<td>A transaction-to-signal transactor for a transaction slave port on the master side of a full AHB bus</td>
</tr>
<tr>
<td>AHB_Master_FS2T</td>
<td>A signal-to-transaction transactor for a transaction master port on the master side of a full AHB bus</td>
</tr>
<tr>
<td>AHB_Lite_Slave_FT2S</td>
<td>A transaction-to-signal transactor for a transaction slave port on the slave side of an AHB-Lite bus</td>
</tr>
<tr>
<td>AHB_Lite_Slave_FS2T</td>
<td>A signal-to-transaction transactor for a transaction master port on the slave side of an AHB-Lite bus</td>
</tr>
<tr>
<td>AHB_Lite_Master_FT2S</td>
<td>A transaction-to-signal transactor for a transaction slave port on the master side of an AHB-Lite bus</td>
</tr>
<tr>
<td>AHB_Lite_Master_FS2T</td>
<td>A signal-to-transaction transactor for a transaction master port on the master side of an AHB-Lite bus</td>
</tr>
</tbody>
</table>

Table 7-1 AHBv2 transactors

The figure below illustrates where each of the transactors should be used with ARM Cycle Model components.

Figure 7-1 AHBv2 transactors
7.1 AHBv2 Transactors and Cycle Model Studio

Cycle Model Studio (CMS) locates SoC Designer AHBv2 transactors from the path pointed to by the $MAXSIM_PROTOCOLS environment variable. If you are working on a CMS project that requires AHBv2 transactors, make sure the AHBv2 Protocol Bundle is installed and set up prior to launching CMS.

8 AHBv2 Port Interfaces

AHBv2 SoC Designer transaction interfaces overcome a problem with the previous version of the AHB interfaces which prohibited a transaction to go through asynchronous paths across a component. The v2 interfaces are described in this chapter.

*Note:* Do not confuse v2 with the AMBA protocol version. v2 refers to the SoC Designer transaction interface version for AMBA AHB, and has no relation to the AMBA protocol specification version number.

8.1 AMBA AHB Interfaces AHBv2 Transaction Ports

AMBA AHB has two distinct interfaces for the master and the slave side of the bus. This is depicted in the figure below.

![AHB Interfaces Diagram](image)

Figure 8-1 AHB Interfaces

AHBv2 ports distinguish between the two distinct AHB interfaces as well as the distinction between the full AMBA AHB and AHB-Lite protocols. The ports are also categorized into transaction master and slave interfaces. This is illustrated in the figures below.
8.1.1 AHB_Master_Port

The ports that face outwards from a component are categorized as AHB transaction master ports. AHB transaction master port, AHB_Master_Port, is defined in the header file AHB_Master_Port.h. AHB master ports are templatized based on the bus type (Full vs. Lite) and the AHB interface (Master side interface vs. Slave side interface), and inherit from the base class, AHB_Master_PortBase.
### 8.1.1.1 AHB_Master_PortBase

```cpp
class WEXP_PORT AHB_Master_PortBase : public MxTransactionMasterPort,
public AHBPortIF
{
    friend class AHB_Slave_PortBase;

public:
    AHB_Master_PortBase(CASIModule* o, std::string n);
    virtual ~AHB_Master_PortBase();

    void setWData(uint32_t data, uint8_t idx = 0);
    uint32_t getRData(uint8_t idx);

    void connect   (CASITransactionIF* iface);
    void disconnect(CASITransactionIF* iface);
    void init(uint32_t addrWidth, uint32_t dataWidth);

private: // disabled methods
    AHB_Master_PortBase();
    AHB_Master_PortBase(const AHB_Master_PortBase&);
    AHB_Master_PortBase& operator=(const AHB_Master_PortBase&);
};
```

**8.1.1.1.1 void setWData(uint32_t data, uint8_t idx = 0)**

Use this method to set HWDATA. This should be called multiple times for data > 32bits, with incrementing `idx`.

**8.1.1.1.2 uint32_t getRData(uint8_t idx);**

Use this method to retrieve HRDATA. This should be called multiple times for data > 32bits, with incrementing `idx`.

**8.1.1.1.3 readDbg/writeDbg**

These methods can be used to initiate debug (zero-cycle) transactions. The desired data size should be encoded into the `ctrl` parameter. Use AHB2_SIZE enum defined in AHB_TLM.h to encode the data size.

**8.1.1.1.4 init(uint32_t addrWidth, uint32_t dataWidth)**

Call this method during the CASI init phase. The default implementation assumes 32 bits for both the address and the data bus.
### 8.1.1.2 AHB_Master_Port Template Specialization

There are four template specializations for AHB master ports:

<table>
<thead>
<tr>
<th>AHB_Master_Port Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;AHBTypeFull, AHBSideMaster&gt;</code></td>
<td>Transaction master port for a full AHB interface, AHB master interface.</td>
</tr>
<tr>
<td><code>&lt;AHBTypeLite, AHBSideMaster&gt;</code></td>
<td>Transaction master port for an AHB-Lite interface, AHB master interface.</td>
</tr>
<tr>
<td><code>&lt;AHBTypeFull, AHBSideSlave&gt;</code></td>
<td>Transaction master port for a full AHB interface, AHB slave interface.</td>
</tr>
<tr>
<td><code>&lt;AHBTypeLite, AHBSideSlave&gt;</code></td>
<td>Transaction master port for an AHB-Lite interface, AHB slave interface.</td>
</tr>
</tbody>
</table>

#### Table 8-1 AHB master port types

### 8.1.1.2.1 AHB_Master_Port<AHBTypeFull, AHBSideMaster>

This is the master port type to use for a port that is on the master side of a full AHB bus.

```cpp
template<>
class WEXP_PORT AHB_Master_Port<AHBTypeFull, AHBSideMaster> : public AHB_Master_PortBase
{
    public:
        AHB_Master_Port(CASIModule* o, std::string n);
    virtual ~AHB_Master_Port() {}  

        void setBusReq(bool busreq);
    void setAddr(uint64_t addr, uint64_t trans, bool write, uint8_t size, uint8_t burst, uint8_t prot, bool lock);

        // AHBPortIF Interface
        uint64_t getSig(AHB2_SIGNAL_IDX sigIdx);
    bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val);
    void clear();
};
```

#### 8.1.1.2.1.1 void setBusReq(bool busreq)

Use this method to set HBUSREQ.

#### 8.1.1.2.1.2 void setAddr(…)

Use this method to set all AHB address and control-related signals.

#### 8.1.1.2.1.3 uint64_t getSig(AHB2_SIGNAL_IDX sigIdx)

This method returns the specified signal value latched during the last CASI communicate phase.
8.1.1.2.1.4  bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val)
Use this method to set individual AHB master signals.

8.1.1.2.2  AHB_Master_Port<AHBTypeLite, AHBSideMaster>
This is the master port type to use for a port that is on the master side of an AHB-Lite bus.

```cpp
template<>
class WEXP_PORT AHB_Master_Port<AHBTypeLite, AHBSideMaster> : public AHB_Master_PortBase
{
    public:
        AHB_Master_Port(CASIModule* o, std::string n);
        virtual ~AHB_Master_Port() {};

        void setAddr(uint64_t addr, uint64_t trans, bool write, uint8_t size, uint8_t burst, uint8_t prot, bool lock);

        // AHBPortIF Interface
        uint64_t getSig(AHB2_SIGNAL_IDX sigIdx);
        bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val);
        void clear();
};
```

8.1.1.2.2.1  void setAddr(…)
Use this method to set all AHB address and control related signals.

8.1.1.2.2.2  uint64_t getSig(AHB2_SIGNAL_IDX sigIdx)
This method returns the specified signal value latched during the last CASI communicate phase.

8.1.1.2.2.3  bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val)
Use this method to set individual AHB master signals.

8.1.1.2.3  AHB_Master_Port<AHBTypeFull, AHBSideSlave>
This is the master port type to use for a port that is on the slave side of a full AHB bus.

```cpp
template<>
class WEXP_PORT AHB_Master_Port<AHBTypeFull, AHBSideSlave> : public AHB_Master_PortBase
{
    public:
        AHB_Master_Port(CASIModule* o, std::string n);
        virtual ~AHB_Master_Port() {};

        void setAddr(uint64_t addr, uint64_t trans, bool write, uint8_t size, uint8_t burst, uint8_t prot, bool mastlock, uint8_t master, bool sel, bool ready);

        // AHBPortIF Interface
        uint64_t getSig(AHB2_SIGNAL_IDX sigIdx);
        bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val);
        void clear();
};
```
8.1.1.2.3.1  void setAddr(…)
Use this method to set the AHB address and control related signals.

8.1.1.2.3.2  uint64_t getSig(AHB2_SIGNAL_IDX sigIdx)
This method returns the specified signal value which was latched during the last CASI communicate phase.

8.1.1.2.3.3  bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val)
Use this method to set individual signals on the port.

8.1.1.2.4  AHB_Master_Port<AHBTypeLite, AHBSideSlave>
This is the master port type to use for a port that is on the slave side of an AHB-Lite bus.

```cpp
template<> class WEXP_PORT AHB_Master_Port<AHBTypeLite, AHBSideSlave> : public AHB_Master_PortBase
{
  public:
    AHB_Master_Port(CASIModule* o, std::string n);
  virtual ~AHB_Master_Port() {} }

  void setAddr(uint64_t addr, uint64_t trans, bool write, uint8_t size, uint8_t burst, uint8_t prot, bool lock, bool sel, bool ready);

  uint64_t getSig(AHB2_SIGNAL_IDX sigIdx);
  bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val);
  void clear();
};
```

8.1.1.2.4.1  void setAddr(…)
Use this method to set the AHB address and control related signals.

8.1.1.2.4.2  uint64_t getSig(AHB2_SIGNAL_IDX sigIdx)
This method returns the specified signal value which was latched during the last CASI communicate phase.

8.1.1.2.4.3  bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val)
Use this method to set individual signals on the port.

**Note:** All AHBv2 port headers are located in $MAXSIM_PROTOCOLS/AHBv2/include.
8.1.2 AHB_Slave_Port

The ports going into a component are categorized as AHB transaction slave ports. The AHB transaction slave port, AHB_Slave_Port, is defined in the header file AHB_Slave_Port.h. These slave ports inherit from the base class, AHB_Slave_PortBase, and are divided into special templatized classes based on the AHB bus and the interface type.

8.1.2.1 AHB_Slave_PortBase

class WEXP_PORT AHB_Slave_PortBase : public CASITransactionSlave,
public AHBPortIF
{
    friend class AHB_Master_PortBase;

public:
    AHB_Slave_PortBase(CASIModule* o, std::string n);
    virtual ~AHB_Slave_PortBase();
    void setRData(uint32_t data, uint8_t idx = 0);
    uint32_t getWData(uint8_t idx);
    void init(uint32_t addrWidth, uint32_t dataWidth);

private: // disabled methods
    AHB_Slave_PortBase();
    AHB_Slave_PortBase(const AHB_Slave_PortBase&);
    AHB_Slave_PortBase& operator=(const AHB_Slave_PortBase&);

    CASIMemoryMapConstraints puMemoryMapConstraints;

public:
    /* debug accesses */
    virtual eslapi::CASIStatus readDbg(uint64_t addr, uint32_t* value,
    uint32_t* ctrl);
    virtual eslapi::CASIStatus writeDbg(uint64_t addr, uint32_t* value,
    uint32_t* ctrl);

    /* Memory map functions */
    virtual int getNumRegions();
    virtual void getAddressRegions(uint64_t* start, uint64_t* size,
    string* name);
    virtual void setAddressRegions(uint64_t* start, uint64_t* size,
    string* name);
    virtual CASIMemoryMapConstraints* getMappingConstraints();

    //**********************************************************
    //*********** IGNORE EVERYTHING FROM THIS LINE *************
    //**********************************************************

public:
    /* Synchronous access functions */
    virtual eslapi::CASIStatus read(uint64_t addr, uint32_t* value,
    uint32_t* ctrl);
    virtual eslapi::CASIStatus write(uint64_t addr, uint32_t* value,
    uint32_t* ctrl);
/* Asynchronous access functions */
virtual eslapi::CASISStatus readReq(uint64_t addr, uint32_t* value,
uint32_t* ctrl,
   CASITransactionCallbackIF* callback);  
virtual eslapi::CASISStatus writeReq(uint64_t addr, uint32_t* value,
uint32_t* ctrl,
   CASITransactionCallbackIF* callback);

/* Arbitration functions */
virtual eslapi::CASIGrant requestAccess(uint64_t addr);
virtual eslapi::CASIGrant checkForGrant(uint64_t addr);

/* CASI : new shared-memory based asynchronous transaction
functions */
virtual void cancelTransaction(CASITransactionInfo* info);  
virtual eslapi::CASISStatus debugTransaction(CASITransactionInfo*
info);

//***********************************************************
//*********** IGNORE EVERYTHING TO THIS LINE ****************
//***********************************************************
};

8.1.2.1  void setRData(uint32_t data, uint8_t idx = 0);
Use this method to set HRDATA. This must be called multiple times with incrementing idx for data greater than 32bits.

8.1.2.1.2  uint32_t getWData(uint8_t idx);
Use this method to retrieve HWDATA. This must be called multiple times with incrementing idx for data greater than 32bits.

8.1.2.1.3  void init(uint32_t addrWidth, uint32_t dataWidth);
This should be called from the CASI init phase to initialize the address and data bus widths.
The default implementation assumed 32 bits for both address and data.

8.1.2.1.4  readDbg/writeDbg
These methods are used for debug (zero-cycle) accesses. These methods should be implemented by the port owner to support debug accesses from the connected masters. The ctrl parameter has an encoded value of the data size to be transferred. This encoding uses the AHB2_SIZE enum defined in AHB_TLM.h.
8.1.2.1.5 Memory map functions

The following methods need to be implemented for memory mapping. Refer to the *SoC Designer ESL API Guide* for information regarding these methods.

- virtual int getNumRegions();
- virtual void getAddressRegions(uint64_t* start, uint64_t* size, string* name);
- virtual void setAddressRegions(uint64_t* start, uint64_t* size, string* name);
- virtual CASIMemoryMapConstraints* getMappingConstraints();

8.1.2.2 AHB_Slave_Port Template Specialization

There are four template specializations for AHB slave ports:

<table>
<thead>
<tr>
<th>AHB_Slave_Port Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;AHBTypeFull, AHBSideMaster&gt;</td>
<td>Transaction slave port for a full AHB interface, AHB master interface.</td>
</tr>
<tr>
<td>&lt;AHBTypeLite, AHBSideMaster&gt;</td>
<td>Transaction slave port for an AHB-Lite interface, AHB master interface.</td>
</tr>
<tr>
<td>&lt;AHBTypeFull, AHBSideSlave&gt;</td>
<td>Transaction slave port for a full AHB interface, AHB slave interface.</td>
</tr>
<tr>
<td>&lt;AHBTypeLite, AHBSideSlave&gt;</td>
<td>Transaction slave port for an AHB-Lite interface, AHB slave interface.</td>
</tr>
</tbody>
</table>

Table 8-2 AHB slave port types

8.1.2.2.1 AHB_Slave_Port<AHBTypeFull, AHBSideMaster>

This is the slave port type to use for a port that is on the master side of a full AHB bus.

template<class WEXP_PORT AHB_Slave_Port<AHBTypeFull, AHBSideMaster> : public AHB_Slave_PortBase
{
    public:
        AHB_Slave_Port(CASIModule* o, std::string n);
        virtual ~AHB_Slave_Port() {};

        void setGrant(bool grant);
        void setResponse(bool ready, uint8_t resp);

        //////////////////////////////////////////////////////////////////////////////////
        // AHBPortIF Interface
        //////////////////////////////////////////////////////////////////////////////////
        uint64_t getSig(AHB2_SIGNALIDX sigIdx);
        bool setSig(AHB2_SIGNALIDX sigIdx, uint64_t val);
        void clear();
};

8.1.2.1.1  void setGrant(bool grant)
Use this method to set \texttt{HGRANT}.

8.1.2.1.2  void setResponse(bool ready, uint8_t resp)
Use this method to set \texttt{HREADY} and \texttt{HRESP}.

8.1.2.1.3  uint64_t getSig(AHB2\_SIGNAL\_IDX sigIdx)
Use this method to retrieve individual AHB signals.

8.1.2.1.4  bool setSig(AHB2\_SIGNAL\_IDX sigIdx, uint64_t val)
Use this method to set individual AHB signals.

8.1.2.2  AHB\_Slave\_Port\langle AHBTyple\_Lite, AHBSide\_Master\rangle
This is the slave port type to use for a port that is on the master side of an AHB-Lite bus.

\begin{verbatim}
    template<>
    class WEXP\_PORT AHB\_Slave\_Port\langle AHBTyple\_Lite, AHBSide\_Master\rangle : public
        AHB\_Slave\_PortBase
        {
            public:
                AHB\_Slave\_Port(CASIModule* o, std::string n);
            virtual ~AHB\_Slave\_Port() {}  
                
                void setResponse(bool ready, uint8_t resp);

                // AHBPortIF Interface
                
                uint64_t getSig(AHB2\_SIGNAL\_IDX sigIdx);
                bool setSig(AHB2\_SIGNAL\_IDX sigIdx, uint64_t val);
                void clear();
        }
\end{verbatim}

8.1.2.2.1  void setResponse(bool ready, uint8_t resp)
Use this method to set \texttt{HREADY} and \texttt{HRESP}.

8.1.2.2.2  uint64_t getSig(AHB2\_SIGNAL\_IDX sigIdx)
Use this method to retrieve individual AHB signals.

8.1.2.2.3  bool setSig(AHB2\_SIGNAL\_IDX sigIdx, uint64_t val)
Use this method to set individual AHB signals.
8.1.2.2.3  AHB_Slave_Port<AHBTypeFull, AHBSideSlave>

Use this slave port type for a port that is on the slave side of a full AHB bus.

```cpp
template<>
class WEXP_PORT AHB_Slave_Port<AHBTypeFull, AHBSideSlave> : public AHB_Slave_PortBase
{
    public:
    AHB_Slave_Port(CASIModule* o, std::string n);
    virtual ~AHB_Slave_Port() {}

    void setResponse(bool readyout, uint8_t resp);

    // AHBPortIF Interface
    uint64_t getSig(AHB2_SIGNAL_IDX sigIdx);
    bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val);
    void clear();
};
```

8.1.2.2.3.1  void setResponse(bool readyout, uint8_t resp)

Use this method to set HREADYOUT and HRESP.

8.1.2.2.3.2  uint64_t getSig(AHB2_SIGNAL_IDX sigIdx)

Use this method to retrieve individual AHB signals.

8.1.2.2.3.3  bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val)

Use this method to set individual AHB signals.

8.1.2.2.4  AHB_Slave_Port<AHBTypeLite, AHBSideSlave>

Use this slave port type for a port that is on the slave side of an AHB-Lite bus.

```cpp
template<>
class WEXP_PORT AHB_Slave_Port<AHBTypeLite, AHBSideSlave> : public AHB_Slave_PortBase
{
    public:
    AHB_Slave_Port(CASIModule* o, std::string n);
    virtual ~AHB_Slave_Port() {}

    void setResponse(bool readyout, uint8_t resp);

    // AHBPortIF Interface
    uint64_t getSig(AHB2_SIGNAL_IDX sigIdx);
    bool setSig(AHB2_SIGNAL_IDX sigIdx, uint64_t val);
    void clear();
};
```
8.1.2.2.4.1  void setResponse(bool readyout, uint8_t resp)
Use this method to set HREADYOUT and HRESP.

8.1.2.2.4.2  uint64_t getSig(AHB2_SIGNAL_ID sigIdx)
Use this method to retrieve individual AHB signals.

8.1.2.2.4.3  bool setSig(AHB2_SIGNAL_ID sigIdx, uint64_t val)
Use this method to set individual AHB signals.

8.1.3  AHBPortIF
AHBPortIF is an abstract class for all AHBv2 port classes. AHBv2 port classes contain two sub-
ports (AHB_Sender_Port and AHB_Receiver_Port) which are not visible in SoC Designer
Canvas or Simulator. These sub-ports enable bi-directional communication between two AHB
ports.
The underlying CASI communication method used behind AHBv2 is driveTransaction, but it
is not recommended that users directly implement driveTransaction, as the AHB_Master_Port
and AHB_Slave_Port port classes already contain methods for the user to model cycle accurate
AHB transactions. The underlying driveTransaction is only used to transfer the AHB signals
set by the port classes from one component to another.

8.1.4  Transaction Phases
AHBv2 relies on a two-phased communication mechanism. The CASI communicate phase is
used only for the transfer of data, and the update phase is used for latching the data and updating
the internal state machine.

<table>
<thead>
<tr>
<th>CASI Phase</th>
<th>Description</th>
</tr>
</thead>
</table>
| communicate | Call sendDrive() to transfer the AHB signals set during the last
              update phase. sendDrive() sends out the buffered data which are
              the updated signals since the last sendDrive() call. |
| update     | Use getSig() to latch the data driven during the current
              communicate phase. setSig() can be used to set the new data to be
              driven out in the next communicate phase. |

Table 8-3  AHBv2 transaction phases

8.1.5  Examples
Source code examples are included in the protocol bundle installation. See
$MAXSIM_PROTOCOLS/AHBv2/src/AHBv2_Master for an example implementation of an AHB
master, and $MAXSIM_PROTOCOLS/AHBv2/src/AHBv2_Slave for an example of an AHB slave
model.
8.2 AHB Extensions

AHBv2 defines signals that are reserved for supporting the additional AMBA2 extensions for specific ARM cores. This section documents the mapping between these extensions and the sideband signals defined in AHBv2. This section describes the `AHB2_SIGNAL_IDX` enumerations declared in `$MAXSIM_PROTOCOLS/AHBv2/include/AHB_TLM.h`.

8.2.1 Cortex-M3 Sideband Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th><code>AHB2_SIGNAL_IDX</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>EXREQ</td>
<td>SIDEBAND0</td>
</tr>
<tr>
<td>MEMATTR</td>
<td>SIDEBAND1</td>
</tr>
<tr>
<td>EXRESP</td>
<td>SIDEBAND2</td>
</tr>
</tbody>
</table>

8-4 Cortex-M3 AHB Extensions