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 2016</td>
<td>A</td>
<td>Non-Confidential</td>
<td>Release with 8.3</td>
</tr>
<tr>
<td>May 2016</td>
<td>B</td>
<td>Non-Confidential</td>
<td>Release with 8.4</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”.

Non-Confidential
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 Suggested Reading ................................................................................................. 6

4 Restrictions and Limitations ................................................................................ 6

5 Bundle Contents .................................................................................................... 6

6 Models ...................................................................................................................... 7
   6.1 CHIToAXI4* ........................................................................................................ 8
   6.2 CHI_to_CHI* ..................................................................................................... 9
   6.3 CHI_Stub ........................................................................................................... 10
       6.3.1 CHI_Stub Macros .................................................................................... 11

7 Probes ...................................................................................................................... 11
   7.1 Tracer Probe .................................................................................................... 11
   7.2 Breakpoint Probe ............................................................................................ 12
   7.3 Monitor Probe .................................................................................................. 13

8 AMBA CHI Port Interfaces ..................................................................................... 14
   8.1 Port Classes ...................................................................................................... 14
       8.1.1 Class Derivations ..................................................................................... 14
   8.2 Node Interface .................................................................................................. 15
       8.2.1 Link Layer Virtual Channels ..................................................................... 15
       8.2.2 Signaling interface .................................................................................... 15
       8.2.3 Interface Methods .................................................................................... 16
           8.2.3.1 Methods for Driving Transactions ...................................................... 16
           8.2.3.2 Notification Handler Methods ............................................................ 16
           8.2.3.3 Transmit Request Methods ............................................................... 16
8.2.3.4 Transmit Response Methods ................................................................. 17
8.2.3.5 Transmit Snoop Methods ....................................................................... 17
8.2.3.6 Transmit Data Methods .......................................................................... 17
8.2.3.7 Methods for Setting Receiver LCredit Signals ......................................... 17
8.2.3.8 Methods for Getting and Setting Signal Values ....................................... 18
8.2.3.9 Methods for Getting and Setting Control Signals .................................... 18
8.2.3.10 Methods for Clearing Signals ............................................................... 18
8.2.3.11 Methods related to Connect and Disconnect .......................................... 18
8.2.3.12 Methods for Address and Bit Width Configuration .............................. 19
8.2.3.13 Reset Methods ..................................................................................... 19

9 Interface Supporting Full System Coherent Memory Views .......... 19
  9.1 Master Port Example Implementation .......................................................... 19
  9.2 Slave Port Example Implementation ............................................................ 20

10 Interface Supporting Fast Debug Access .............................................. 21
  10.1 Slave Port Example Implementation .......................................................... 21
1 Introduction

This is the user guide for the SoC Designer Plus AMBA CHI Protocol Bundle. This protocol bundle contains the SoC Designer Plus transaction port interface for the ARM AMBA CHI protocol.

The ARM AMBA CHI protocol represents a paradigm shift with respect to interface composition. AMBA CHI operates on the concept of Nodes and Interfaces, rather than the Master/Slave paradigm used by previous AMBA protocols. AMBA CHI nodes and interfaces signals are usually mirror-images of each other (exceptions are noted), and the methods supported are the same for both.

For more information, refer to the ARM AMBA5 CHI Architecture Specification.

2 Requirements

The AMBA CHI protocol bundle requires the following:

- SoC Designer Plus v8.3 or later
- Compilation tools as set forth in the SoC Designer Plus Installation Guide.

3 Suggested Reading

- SoC Designer Plus User Guide
- SoC Designer Plus Installation Guide
- SoC Designer Plus System Analyzer User Guide
- SoC Designer Plus System Analyzer API Reference
- ARM AMBA5 CHI Architecture Specification

4 Restrictions and Limitations

- Monitored CHI data is viewable only with the System Analyzer; it is not visible in the SoC Designer Plus monitoring views. Use SoC Designer Plus to create monitors and run simulations; during simulation, monitored data is exported to the System Analyzer database. See the SoC Designer Plus System Analyzer User Guide for more information.
- CHI Profiling is not currently supported.
- CHI Debug Transactions are not currently supported.

5 Bundle Contents

This bundle contains protocol support packages for the ARM AMBA CHI protocol, a component that converts between CHI and AXI4, a scriptable CHI master component. Also included are probes, which provide visibility into transactions between components, and example models.
## 6 Models

Table 6-1 lists the components included in this bundle. These are described in more detail throughout this section.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIToAXI4*</td>
<td>Converts CHI traffic to AXI4 format. Note the following: <strong>• CHI</strong> sets Address Width to 44 bits. <strong>• CHI</strong> RSVDC Width is set to 4 bits.</td>
</tr>
</tbody>
</table>
| CHI_To_CHI*       | Updates the RSVDC field in REQFLIT as follows: **• REQFLIT** RSVDC[3:2] = Cluster_ID  **• REQFLIT** RSVDC[1:0] = LPID [1:0] with the following:  
|                   |   **a.** LPID[2:0]=Logical Processor ID field read from **REQFLIT**[93:91].  **b.** Cluster_ID is a 2-bit integer that represents the CPU cluster  
|                   |   connected to component’s slave port.  
|                   | Note: If RSVDC is larger than 4 bits for a system, those bits are set to 0.                                                                                                                                   |
| CHI_Strip         | A scriptable CHI master component.                                                                                                                                                                            |
| CHI_LinkRequester | Example models that you can use to build your own CASI models using the API.                                                                                                                                     |
| CHI_LinkSlave     |                                                                                                                                                                                                           |

### Table 6-1 CHI Bundle Components

The .conf files for CHI components are located at:

- $MAXSIM_HOME//Protocols/CHI/etc/CHIComponent.conf
- $MAXSIM_HOME//Protocols/CHI/etc/CHIProbe.conf (include for waveform support)
- $MAXSIM_HOME//Protocols/TLM2/amba_socket/etc/AMBAComponents.conf (CHI_to_CHI)
6.1 CHIToAXI4*

Table 6-2 lists the ports for the CHIToAXI4* component.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axi4*_m</td>
<td>Transaction Master</td>
</tr>
<tr>
<td>chi_s</td>
<td>Transaction Slave</td>
</tr>
<tr>
<td>clk-in</td>
<td>Clock Slave</td>
</tr>
</tbody>
</table>

Table 6-2 CHIToAXI4* Ports

Table 6-3 lists the parameters for the CHIToAXI4* component.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHI Data Width</td>
<td>Width in bits of the CHI data bus. Accepted values are 128, 256, and 512. The default is 128.</td>
</tr>
<tr>
<td>AXI4 Data Width</td>
<td>Width in bits of the AXI4 data bus. Accepted values are 32, 64, 128, 256, and 512. The default is 128.</td>
</tr>
<tr>
<td>Enable Debug Messages</td>
<td>When set to true, the model debug messages are displayed as output.</td>
</tr>
<tr>
<td>CHI Protocol Variant</td>
<td>Sets the protocol for the chi_s port. Select from the following options: CHI-SNF and CHI-SNI.</td>
</tr>
<tr>
<td>AXI4 Protocol Variant</td>
<td>Sets the protocol for the axi4*_m port. Select from the following options: AXI4 and ACE-Lite.</td>
</tr>
</tbody>
</table>

Table 6-3 CHIToAXI4* Parameters
6.2 **CHI_to_CHI**

This component supports systems that have one or two clusters, up to eight CPUs, and include GIC-400 controllers. The CHI_to_CHI* component sets bits in the RSVDC field in REQFLIT as described in *Section 6, Models*.

Table 6-2 lists the ports for the CHI_to_CHI* component.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chi_m</td>
<td>Transaction Master</td>
</tr>
<tr>
<td>chi_s</td>
<td>Transaction Slave</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Width</td>
<td>Width in bits of the CHI data bus. Accepted values are 128, 256, and 512. The default is 128.</td>
</tr>
<tr>
<td>Cluster ID</td>
<td>A 2-bit integer signifying the CPU-cluster connected to this component’s slave port (chi_s).</td>
</tr>
<tr>
<td>Enable Debug Messages</td>
<td>When set to true, the model debug messages are displayed as output.</td>
</tr>
<tr>
<td>Protocol Variant</td>
<td>Sets the CHI protocol for the component. Select from the following options: CHI-SNF, CHI-SNI, CHI-RNF, CHI-RND, and CHI-RNI. Default is CHI-SNF.</td>
</tr>
</tbody>
</table>

**Table 6-2 CHI_to_CHI* Ports**

**Table 6-3 CHI_to_CHI* Parameters**
### 6.3 CHI_Stub

CHI_Stub is a master component which can be controlled with a SoC Designer `mxscr` script. Table 6-4 lists the ports for the CHI*_Stub component.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chi_m</td>
<td>Transaction Master port</td>
</tr>
<tr>
<td>p_in[0-3]</td>
<td>Signal Slave ports</td>
</tr>
<tr>
<td>p_out[0-3]</td>
<td>Signal Master ports</td>
</tr>
<tr>
<td>clk_in</td>
<td>Clock Slave port</td>
</tr>
<tr>
<td>reset</td>
<td>Executes a reset (RESET_SOFT) on the stub component.</td>
</tr>
</tbody>
</table>

Table 6-4 CHI_Stub Ports

Table 6-5 lists the component parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amba_name</td>
<td>These parameters are obsolete and should be left at their default values. It is recommended that you use the Memory Map Editor (MME) in SoC Designer Plus, which provides centralized viewing and management of the memory regions available to the components in a system. For information about migrating existing systems to use the MME, refer to the SoC Designer Plus User Guide.</td>
</tr>
<tr>
<td>amba_size</td>
<td></td>
</tr>
<tr>
<td>amba_start</td>
<td></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>Width in bits of the data bus. It must match the data bus width of the connected model. Allowed values are 128, 256, and 512.</td>
</tr>
<tr>
<td>Enable Debug Messages</td>
<td>When set to true, the model debug messages are displayed as output.</td>
</tr>
<tr>
<td>Protocol Variant</td>
<td>Select from the following options: ACE, ACE-Lite+DVM, ACE-Lite, AXI4, AXI4-Lite, RNF, RNI, and RND.</td>
</tr>
</tbody>
</table>

Table 6-5 CHI*_Stub Parameters
6.3.1 CHI_Stub Macros

Macro definitions for CHI_Stub are provided in the following files:

$MAXSIM_PROTOCOLS/CHI/include/CHI_Stub_Macros.h
$MAXSIM_PROTOCOLS/CHI/include/CHI_Stub_CheckMacros.h
$MAXSIM_PROTOCOLS/CHI/include/AMBA_PVE_Stub_CheckMacros.h
$MAXSIM_PROTOCOLS/CHI/include/AMBA_PVE_Stub_Macros.h

For information about how to use the macros, read the comments in the .h files.

7 Probes

The following simulation probes are included in this Protocol Bundle.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHI Tracer</td>
<td>Enables tracing of CHI signals. You can view traced signals in the SoC Designer Plus simulator waveform window.</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Transaction breakpoint on a CHI connection.</td>
</tr>
<tr>
<td>Monitor</td>
<td>Monitors activity over a CHI connection for each cycle. Results can be viewed using System Analyzer.</td>
</tr>
</tbody>
</table>

Table 7-1 Probes

7.1 Tracer Probe

This probe allows tracing of CHI signals. On the tracer, the FLIT signals are decoded into individual fields. You can view traced signals in the SoC Designer Plus waveform window. To add a tracer probe, right-click on a CHI connection and select “Enable/Disable Tracing.” This displays the Tracer Properties dialog (see the SoC Designer User Guide for more information about the Tracer Properties dialog).

By default, all signals are traced. This can be changed by using the checkboxes located on the left side of the signal.
7.2 Breakpoint Probe

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 will break on any active CHI transaction across the connection.

Channel and Signal options differ depending on the protocol variant of the connection.

You can create multiple breakpoints on multiple channels. To configure breakpoint conditions, display the breakpoint property dialog by right-clicking on the connection and selecting “Edit Breakpoint Properties.” Figure 7-1 shows the CHI breakpoint condition dialog.

![Figure 7-1 CHI Breakpoint Condition dialog](image)

Breakpoint condition options are as follows:

**Breakpoints panel:**
- **Breakpoint selection** menu – Select the desired breakpoint to display its properties in the Signal panel.
- **Enabled** checkbox – Select to keep the breakpoint enabled (this is the default), or deselect to disable the breakpoint temporarily. Disabling does not delete the breakpoint.
- **Channel selection** menu – Select the channel you want to set the breakpoint on. Channel selections differ according to the protocol setting on the component.
- **Activity selection** menu – Select Any Activity, OR signals, or AND signals for this breakpoint.
- **Add** button – Click to create a new breakpoint. A new number appears in the Breakpoint selection menu and the Signals panel clears so you can specify its properties.
- **Delete** button – Deletes the selected breakpoint.
Signals panel: Use this panel to set the conditions for the breakpoint you’re defining.
- **Enabled** checkbox - Select to keep the breakpoint for this condition enabled (this is the default), or deselect to disable it. Disabling does not delete the condition.
- **Signal** menu – Select the signal you want to break on.
- **Value 1** and **Value 2** fields – Enter specific values as needed, for example, when setting a breakpoint on a particular address or within a specific range.
- **Symbol** field –
- **Delete Signal** button – Deletes the associated signal breakpoint.

### 7.3 Monitor Probe

The Monitor probe enables monitoring of per-cycle activity on a CHI connection.

To enable a monitor probe:

1. Right-click on a connection.
2. From the context menu, select **Insert/Remove Monitor**.

During simulation, monitored content is exported to the System Analyzer database. Refer to the *SoC Designer System Analyzer User Guide* for information about viewing monitored activity.
8 AMBA CHI Port Interfaces

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

AMBA CHI port classes are CASI implementations of the ARM AMBA CHI protocol. The interfaces for AMBA CHI transactions are described in this chapter.

8.1 Port Classes

The port class header files are located under the $MAXSIM_PROTOCOLS/AMBA CHI/include directory. These header files are needed for building SoC Designer Plus components with AMBA CHI ports. All interface classes are under C++ namespace casichi.

8.1.1 Class Derivations

The CHI_Requester_Port class provides the interface for CHI requester ports. CHI requester ports must derive from this class to work with the CHI specification.

The CHI_Slave_Port class provides the interface for CHI slave ports. CHI slave ports must derive from this class to work with the CHI specification.

Interconnect ports have not been specifically defined but must be derived from the corresponding class to which they are paired. If an interconnect port is connected to a CHI requester port, it must be derived from the CHI_Slave_Port class. Likewise, if an interconnect port is connected to a CHI slave port, it must be derived from the CHI_Requester_Port class.
8.2 Node Interface

Nodes communicate by exchanging link layer flits using the node interface. A Flit is the basic unit of transfer in the link layer. Packets are formatted into flits and transmitted across links.

8.2.1 Link Layer Virtual Channels

The following figure shows the virtual transmit and receive channels on the link layer.

![Link Layer Virtual Channels Diagram]

A specific node port acts as both a transmitter and a receiver depending on the type of interface it supports (see Section 6.2.2, Signaling Interface).

8.2.2 Signaling interface

The following table describes the transactions supported by each node interface.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
<th>Virtual Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN-F</td>
<td>Used by fully coherent request nodes such as CPU cores and core clusters.</td>
<td>TX → TX → RX ← RX ← TX → RX ←</td>
</tr>
<tr>
<td>RN-I</td>
<td>Used by IO coherent nodes such as GPU and IO bridges.</td>
<td>TX → TX → RX ← RX ←</td>
</tr>
<tr>
<td>RN-D</td>
<td>Used by IO coherent nodes that process</td>
<td>TX → TX → TX →</td>
</tr>
</tbody>
</table>

ARM DUI 0954B Copyright© 2016 ARM Limited. All Rights Reserved
### 8.2.3 Interface Methods

This section describes the methods provided for CHI port interfaces.

#### 8.2.3.1 Methods for Driving Transactions

The following methods are provided for driving flit channel signals. They are called by the node interface with an associated (connected) TX channel and must be implemented by a node interface for cycle based reception of the channel signals.

- virtual void driveTransactionCB_RXCTL() {}; (Receive Control Channel)
- virtual void driveTransactionCB_RXREQ() {}; (Receive Request Channel)
- virtual void driveTransactionCB_RXRSP() {}; (Receive Response Channel)
- virtual void driveTransactionCB_RXSND() {}; (Receive Snoop Channel)
- virtual void driveTransactionCB_RXDAT() {}; (Receive Data Channel)

Refer also to the associated methods in Section 8.2.3.9, Methods for Getting and Setting Control Signals.

- void sendDrive();
  
  Call the sendDrive() method in the component's Communicate phase to automatically call driveTransaction on all the sender ports.

#### 8.2.3.2 Notification Handler Methods

Notify handlers allow reception of reverse direction signals. For CHI, these allow reception of the LCRDV (L-credit) signal. They must be implemented by a node interface for reception of reverse direction channel signals. Note that notifications are only guaranteed when the LCRDV signal changes.

- virtual void notifyEventCB_TXREQ() {};
- virtual void notifyEventCB_TXRSP() {};
- virtual void notifyEventCB_TXSND() {};
- virtual void notifyEventCB_TXDAT() {};

#### 8.2.3.3 Transmit Request Methods

The Transmit Request methods are used for setting the signals on the TXREQ channel only. In most cases, they should only be called during the Update phase. The signals are driven in the subsequent Communicate phase.

- void setTXREQFLITPEND(bool value)
- void setTXREQFLITV(bool value)
8.2.3.4 Transmit Response Methods

The Transmit Response methods are used for setting the signals on the TXSNP channel only. In most cases, they should only be called during the Update phase. The signals are driven in the subsequent Communicate phase.

- void setTXREQFLIT(uint8_t QoS, uint8_t TgtID, uint8_t SrcID, uint8_t TxnID, uint8_t Opcode, uint8_t Size, uint64_t Addr, bool NS, bool LikelyShared, bool DynPCrd, uint8_t Order, uint8_t PCrdType, uint8_t MemAttr, uint8_t SnpAttr, uint8_t LPID, bool Excl, bool ExpCompAck, uint8_t RSVDC);

8.2.3.5 Transmit Snoop Methods

The Transmit Snoop methods are used for setting the signals on the TXSNP channel only. In most cases, they should only be called during the Update phase. The signals are driven in the subsequent Communicate phase.

- void setTXSNPFLITPEND(bool value)
- void setTXSNPFLITV(bool value)
- void setTXSNPFLIT(uint8_t QoS, uint8_t TgtID, uint8_t SrcID, uint8_t TxnID, uint8_t Opcode, uint8_t RespErr, uint8_t Resp, uint8_t DBID, uint8_t PCrdType);

8.2.3.6 Transmit Data Methods

Transmit Data methods are used for setting the signals on the TXDAT channel only. In most cases, they should only be called during the Update phase. The signals are driven in the subsequent Communicate phase.

- void setTXDATFLITPEND(bool value)
- void setTXDATFLITV(bool value)
- void setTXDATFLIT(uint8_t QoS, uint8_t TgtID, uint8_t SrcID, uint8_t TxnID, uint8_t Opcode, uint8_t RespErr, uint8_t Resp, uint8_t DBID, uint8_t CCID, uint8_t DataID, uint8_t RSVDC, uint64_t BE);
- void setTXDATFLITData(uint32_t data, uint8_t idx = 0);

8.2.3.7 Methods for Setting Receiver LCredit Signals

The LCredit methods are used for setting the LCredit signals on the RX channels. In most cases, they should only be called during the Update phase. The signals are driven in the subsequent Communicate phase.

- void setRXREQLCRDV(bool value);
- void setRXSPLCRDV(bool value);
- void setRXSNPCLRDV(bool value);
- void setRXDATLCRDV(bool value);
8.2.3.8 Methods for Getting and Setting Signal Values

- **bool setSig(CHI_CHANNEL chnldx, uint8_t sigIdx, uint32_t val)**
  You can use `setSig` to set one value to one signal on any channel.

  **Note:** You can only set signals that are generated by the Requester port. Signals generated by the slave port are ignored.

- **uint32_t getSig(CHI_CHANNEL chnldx, uint8_t sigIdx);**
  You can use `getSig` at any time to see the current signal values for any channel.

  **Note:** Any set method calls performed in the same phase as `getSig` are not reflected, because the current signals are updated in the subsequent Communicate phase.

- **uint32_t getRXDATFLITData(uint8_t idx);**
  `getRXDATFLITData` is the same as `getSig`, but provides an easier way to retrieve a particular byte of data on the RXDAT channel. `idx` points to the 32-bit data word being referenced.

8.2.3.8.1 chnlIdx/sigIdx values

The `chnlIdx` values are of type `CHI_CHANNEL`.

The `sigIdx` values are of type `CHI_<channel>_SIGNAL_INDEX` (where `channel` is REQ, RSP, SNP, or DAT). These types are defined in the provided include file `CHI_TLM.h`.

8.2.3.9 Methods for Getting and Setting Control Signals

The following are convenience methods for Control signals:

- **void setTXSACTIVE(bool val);**
- **void setTXLINKACTIVEREQ(bool val);**
- **void setRXLINKACTIVEACK(bool val);**
- **bool getTXSACTIVE();**
- **bool getTXLINKACTIVEREQ();**
- **bool getRXLINKACTIVEACK();**
- **bool getRXLINKACTIVEACK();**

8.2.3.10 Methods for Clearing Signals

`clear` is a convenience function that clears all signals on all channels. It should only be called during Update:

- **void clear();**

8.2.3.11 Methods related to Connect and Disconnect

Connect and disconnect are used during the interconnect phase in SoC Designer Plus and are utilized by the SoC Designer Simulator. For normal usage, these functions do not need to be called by a user (client) of the node interface classes.

- **void connect(CASITransactionIF* iface);**
8.2.3.12 Methods for Address and Bit Width Configuration

Call this method during Initialization to set up the data bit width, protocol ID, and protocol Name of the CHI master port:

- `void init(uint32_t dataWidth, uint32_t protocolID, const string protocolName);`

The protocolName must be “CHI”. The protocolID must be one of the following constants defined by the node interface (the requester and slave nodes must match):

- CHI_RNF_PROTOCOL_ID
- CHI_RNI_PROTOCOL_ID
- CHI_RND_PROTOCOL_ID
- CHI_SNF_PROTOCOL_ID

8.2.3.13 Reset Methods

Call these methods during Reset to ensure proper state before running again:

- `void reset();`
- `virtual bool saveData(eslapi::CASIODataStream& os);`
- `virtual bool restoreData(eslapi::CASIIDataStream& is);`

9 Interface Supporting Full System Coherent Memory Views

To support full system coherent memory views, the CHI Master and Slave ports implement the interface `DebugReverseIF`, which includes the function `debugTransactionReverse`. This function resides in `Protocols/CHI/Include/DebugReverseIF.h`.

Refer to the `SoC Designer Plus User Guide` for more information about full system coherent memory views, and to the `MxScript Reference Manual` for details about how CADIMemWrite and CADIMemRead support full system coherent memory views.

9.1 Master Port Example Implementation

Following is an example class implementation of the CHI_XTOR_TPORT port:

```cpp
#include "CHI_RequesterDefs.h"
#include "CHI_XTOR_TPort.cpp"

eslapi::CASISStatus
CHI_S2T_TM::debugTransactionReverse(eslapi::CASITransactionInfo *info)
{
    if(debugReverseCallback != NULL)
    {
```
return debugReverseCallback(owner_module, info);
}
else{
    return eslapi::CASI_STATUS_NOTSUPPORTED;
}
}

eslapi::CAInterface* CHI_S2T_TM::ObtainInterface(eslapi::if_name_t ifName,
    eslapi::if_rev_t minRev,
    eslapi::if_rev_t* actualRev)
{
    eslapi::CAInterface *intf = MxTransactionMaster::ObtainInterface(ifName,
    minRev, actualRev);
    if(intf == NULL)
    {
        intf = DebugReverseIF::ObtainInterface(ifName, minRev, actualRev);
    }
    return intf;

9.2 Slave Port Example Implementation
Following is an example class implementation of the CHI_XTOR_TPORT port:

#include "CHI_SlaveDefs.h"
#include "CHI_XTOR_TPort.cpp"

eslapi::CASISStatus
CHI_T2S_TS::debugTransactionReverse(eslapi::CASITransactionInfo *info){
    eslapi::if_rev_t actualRev;
    if(getMaster() != NULL)
    {
        DebugReverseIF* master_p = dynamic_cast<DebugReverseIF*>(getMaster()->
>ObtainInterface("DebugReverseIF", 0, &actualRev));
        if(master_p != NULL)
        {
            return master_p->debugTransactionReverse(info);
        }
        else
        {
            return eslapi::CASI_STATUS_NOTSUPPORTED;
        }
    }else{
        return eslapi::CASI_STATUS_NOTSUPPORTED;
    }
}
eslapi::CAInterface* CHI_T2S_TS::ObtainInterface(eslapi::if_name_t ifName,
eslapi::if_rev_t minRev,
eslapi::if_rev_t* actualRev)
{
    eslapi::CAInterface *intf = CASITransactionSlave::ObtainInterface(ifName,
    minRev, actualRev);
    if(intf == NULL)
    {
        intf = DebugReverseIF::ObtainInterface(ifName, minRev, actualRev);
    }
    return intf;
}

10 Interface Supporting Fast Debug Access

Fast debug access is a form of debug access which ignores the BUS width. This is mainly used
for loading larger application images into CPUs.

To support fast debug access, the CHI Master and Slave ports implement the interface
fast_debug_access_if, which includes the function debugTransactionReverse. This function
resides in MaxSim/eslapi/CASIInterfaces.h.

10.1 Slave Port Example Implementation

The fast_debug_access_if interface is implemented on slave ports only. The following is an
example implementation:

#include "CHI_SlaveDefs.h"
#include "CHI_XTOR_TPort.cpp"
#include "maxsimCompatibility.h"

eslapi::CASIStatus
CHI_XTOR_TPORT::debugTransaction(eslapi::CASIDebugTransactionInfo *info)
{
    if (owner->mfDbACb.fastDebugAccess != NULL)
        return CarbonDebugFunctionsToDebugAccess(owner->mfDbACb, info);
    else
    {
        // Normal debug access
    }
}