Arm® AMBA® TLM 2.0 Library

Developer Guide

arm

101459_1000_00 (ID011519)

Arm AMBA TLM 2.0 Library
Developer Guide

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
January 2019	1000-00	Non-Confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

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

Contents
Arm AMBA TLM 2.0 Library Developer Guide

Preface

This preface introduces the *Arm® AMBA® TLM 2.0 Library Developer Guide.* It contains the following sections:

About this book

This developer guide describes a C++ library of *Transaction-Level Modeling* (TLM) 2.0-compatible type definitions for modeling AMBA AXI4 and ACE ports on SystemC models with approximate and cycle-accurate timing requirements.

Intended audience

This document is written for experienced hardware and software developers to aid the development of TLM 2.0 compatible models that communicate over AMBA AXI buses.

You must be familiar with:

- The basic concepts of C++ such as classes and inheritance.
- SystemC and TLM 2.0 standards.
- TLM 2.0 Generic Protocol payloads and phases and TLM 2.0 sockets and the motivations for the form of those mechanisms.
- The general principles of the AMBA AXI.

Using this book

This document is organized into the following chapters:

Chapter 1 *AMBA TLM 2.0 library overview*

Read this for a library overview of the C++ classes.

Chapter 2 *AXI4 protocol*

Read this for a description of the AXI4 Protocol.

Chapter 3 *AXI4 payload*

Read this for a description of the AXI4 Payload.

Glossary

The *Arm® Glossary* is a list of terms used in Arm documentation, together with definitions for those terms. The *Arm® Glossary* does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See *Arm® Glossary* http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

Typographical conventions

The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
monospace	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

monospace italic	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
monospace bold	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This section lists related publications by Arm® and by third parties.

See Infocenter http://infocenter.arm.com for access to Arm documentation.

Arm publications

The book contains information that is specific to this product. The following publications provide reference information about the Arm architecture:

- *Arm® AMBA® AXI Protocol Specification*, *AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite* (ARM IHI 0022).
- *Arm® Architecture Reference Manuals* http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html.

Other publications

This section lists relevant documents published by third parties.

For additional information on the Accellera Systems Initiative, see Accellera http://www.accellera.org.

The following publication provides reference information about the SystemC and TLM 2.0 standards:

- IEEE Std 1666-2011, *IEEE Standard for Standard SystemC Language Reference Manual*, January 2012.

Feedback

Arm welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title.
- The number, 101459_1000_00.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Chapter 1
AMBA TLM 2.0 library overview

This chapter describes a library overview of C++ classes which allow approximately-timed and cycle-accurate AMBA AXI4 and ACE ports to be modeled and simulated as part of SystemC models. It contains the following sections:

1.1 Header file structure

The Arm Performance Modeling TLM library is provided as a pre-compiled binary library and several C++ header files defining:

- Payload types.
- Communication phases.
- Base socket definitions.
- Types to identify the protocol that is used at particular sockets.

Table 1-1 shows the header files provided.

Table 1-1 AXI header files

Header file	Contents
arm_axi4.h	This file includes all the header files that are required to model AMBA AXI4.
arm_axi4_payload.h	This file consists of the constituent types that are required to describe a data transaction and control payload for AXI4. The payload types are described in Chapter 3 *AXI4 payload*.
arm_axi4_phase.h	This file consists of the phase type that is used to describe events in AXI4 for approximately-timed and cycle-accurate modeling. This type is described in the context of the protocols they form in *Phases* on page 2-3.
arm_axi4_socket.h	This file consists of base socket definitions for master and slave TLM 2.0 sockets and AXI ports. These definitions are described in *Socket types* on page 1-3.
arm_tlm_helpers.h	This file consists of miscellaneous helper type definitions that are used by other header files.
arm_tlm_protocol.h	This file consists of an enumeration of the varieties of AXI supportable by a socket.
arm_tlm_socket.h	This file consists of common base socket definitions that contain protocol and port width definitions for a port. However, these base socket definitions are not committed to a particular payload or protocol. These definitions are described in *Socket types* on page 1-3.

1.2 Socket types

Base sockets are provided as the types ARM::TLM::BaseSlaveSocket, derived from tlm::tlm_target_socket and ARM::TLM::BaseMasterSocket, derived from tlm::tlm_initiator_socket. You must derive ports on models from these base sockets.

A single socket models an entire AXI4 port including:

- Address.
- Data.
- Response.
- Snoop.
- ACE **WACK** and **RACK** channels.

ARM::TLM::BaseSlaveSocket and ARM::TLM::BaseMasterSocket are templates that must be specialized with the payload type and protocol-defining phase type of the port.

Table 1-2 shows the supported specializations:

Table 1-2 Supported specializations

Socket types	Applicability
ARM::TLM::BaseSlaveSocket<ARM::AXI4::ProtocolType> ARM::TLM::BaseMasterSocket<ARM::AXI4::ProtocolType>	For AXI4 models

The typesARM::AXI4::BaseSlaveSocket and ARM::AXI4::BaseMasterSocket are provided as a convenience with ARM::AXI4::ProtocolType as their default specialization.

The following subsections describe:

- *Socket protocol and port width*.
- *Binding sockets* on page 1-4.
- *Supported transport interfaces* on page 1-4.
- *Simple sockets* on page 1-5.

1.2.1 Socket protocol and port width

In addition to the port name, the two arguments are passed, on construction. Table 1-3 shows the two arguments that configure the expected communications on the port:

Table 1-3 Configuration argument type

Argument	Type	Meaning
protocol	ARM::TLM::Protocol	A choice of protocol variant that is expected on the port.
port_width	unsigned	The width of the data channels of the port in bits. This width must be a power of two and be valid according to the AXI4 *Architecture* specifications. For more information, see the *Arm® AMBA® AXI Protocol Specification*, *AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*.

Table 1-4 shows the values that the protocol argument can take:

Table 1-4 Protocol values

Protocol value	Short description
PROTOCOL_ACE	Full ACE with *Distributed Virtual Memory* (DVM), barriers, snoop channels and ACE **RACK** and **WACK** channels.
PROTOCOL_ACE_LITE	ACE-Lite. ACE without snoop channels and **RACK** and **WACK** channels.
PROTOCOL_ACE_LITE_DVM	ACE-Lite with extra DVM support.
PROTOCOL_AXI4	AXI4 without any ACE features.
PROTOCOL_AXI4_LITE	AXI4-Lite. AXI with simpler payload options.

1.2.2 Binding sockets

Sockets are bound together as described in the IEEE standard for TLM 2.0 sockets. See IEEE Std 1666-2011, *IEEE Standard for Standard SystemC® Language Reference Manual*, for more information. ARM::TLM::BaseSlaveSocket and ARM::TLM::BaseMasterSocket overload the member function bind() to try and ensure that the protocol and port width of bound sockets exactly match. Connecting sockets with different protocols or port widths results in a SystemC error.

1.2.3 Supported transport interfaces

Only a subset of the interfaces that are provided by tlm::tlm_fw_transport_if and tlm::tlm_bw_transport_if of TLM 2.0 are supported with the protocols described in this document.

The following subsections describe:

- *Non-blocking transport*.
- *Blocking transport*.
- *DMI*.
- *Debug transport*.

Non-blocking transport

All sockets support non-blocking forward and backward calls. They are the standard communication method that is used by all protocols that are listed in this document.

Blocking transport

Blocking transport is not supported by the protocols listed in this document.

DMI

DMI is not supported by the protocols listed in this document.

Debug transport

Sockets must implement the transport_dbg() call. Not all models are capable of dealing with debug transactions correctly and it is always valid for a model to return 0 and so decline to accept debug transactions.

1.2.4 Simple sockets

Sockets ARM::TLM::SimpleMasterSocket and ARM::TLM::SimpleSlaveSocket are provided for convenience when implementing non-blocking transport and debug transport using member functions on objects other than sockets. These work in a similar way to tlm_utils::tlm_simple_initiator_socket and tlm_utils::tlm_simple_target_socket of TLM 2.0. However, they do not provide any blocking to non-blocking adaptation.

Chapter 2
AXI4 protocol

This chapter provides an overview of the AXI4 protocol. This protocol aims to represent the low-level signaling present in the underlying AXI channels of a port. This protocol enables cycle-accurate models of AXI4 port behavior to be constructed.

It contains the following sections:

2.1 Clocking

Non-blocking protocol communications are always associated with a clocking regime. Bound sockets must share a clocking regime and that regime must have two distinct alternating periods in each clock cycle. These periods are:

Communicate

This occurs when the TLM 2.0 interface calls are made between ports.

Update

This occurs when the internal state updates in response to communications.

Two periods are required so that all communications in a cycle can be known to have occurred before a state update at the start of the next cycle takes place. There is no requirement for a model to have a free-running clock signal, but it must respect the Communicate and Update clock periods of the model to which it is connected.

Where a free-running clock signal with alternate positive and negative-going events is present, you can implement this clocking model by using those events for Update and Communicate respectively.

Arm requires that at least where a clock signal associated with a socket is present, then no communication must take place where it is triggered by the positive-going event of that clock. Arm recommends that all communication is triggered by the negative-going edge of the clock.

To this end, this developer guide refers to events causing communication as negedge and those causing state update as posedge.

2.2 Phases

Each AXI4 port logically consists of a number of AXI channels. The complete set of channels when using full ACE ports are:

- AW.
- W.
- B.
- AR.
- R.
- AC.
- CR.
- CD.
- WACK.
- RACK.

All channels of a port are modeled with a single TLM 2.0 socket. Each channel has a pair of phase enumeration values from the type ARM::AXI4::Phase representing **VALID** and **READY** signaling on that channel. For example, AW channel has phases AW_VALID and AW_READY.

With non-blocking communication, AXI channels from master port to slave port, for example, AW, send their VALID event using nb_transport_fw() and receive a READY event either as a phase update to an nb_transport_fw() call or by making an nb_transport_bw() call. AXI channels from slave port to master port, for example, B send VALID with nb_transport_bw() and READY with nb_transport_fw(). Figure 2-1 shows the AXI Channel State Machine.

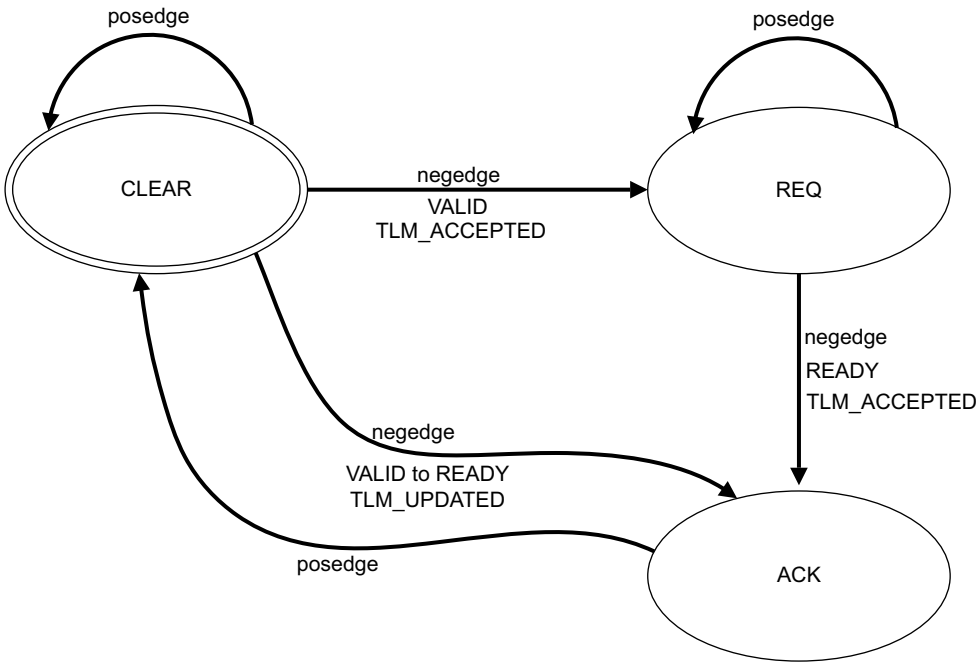

Figure 2-1 AXI Channel State Machine

Each AXI channel, other than WACK and RACK, obeys the state machine that is shown in Figure 2-1 on page 2-3. From state CLEAR, sending a VALID phase is analogous to raising a **VALID** signal on an AXI channel. The port receiving the VALID phase call can then respond with READY, for example, AW_VALID is responded to with AW_READY, which is analogous to the **READY** signal being observed after a raised **VALID** signal.

READY is communicated by either changing the passed phase during the nb_transport_fw and bw() call (phase to READY) and returning with tlm::TLM_UPDATED, or by responding later in the opposite direction with a READY phase. These two options are shown on Figure 2-1 on page 2-3 as transitions from CLEAR to ACK and REQ respectively. Responding immediately with READY indicates that the AXI channel communication has been accepted in the same cycle as the VALID was presented. Responding later can indicate same-cycle READY, where a model can generate the required response function call before posedge, or where the response is in a difference cycle, after the next posedge, it indicates a delayed READY.

Transition from the ACK state back to CLEAR in Figure 2-1 on page 2-3 only happens when the next posedge period passes. This restriction ensures that no more than one handshake takes place on each AXI channel in each clock cycle. Where a free-running clock is not present, this restriction must still be kept. Do this by ensuring that the next update period in the agreed clocking regime has passed before a next VALID phase can be sent.

With reference to AXI4, although the *Arm® AMBA® AXI Protocol Specification, AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*, allows the **READY** signal to rise before the **VALID** signal has arrived, this library does not model this behavior. So a READY call must only be generated as a response to a received VALID phase. A VALID phase therefore effectively indicates that both **READY** and **VALID** signals are HIGH.

Note

In all cases, a READY phase call must reference the same payload as that passed by the VALID phase call.

Table 2-1 lists all phases, the direction they must be transmitted, which phases they must be replied with, optionally immediately using tlm::TLM_UPDATED reply, and the signal-level interpretation of the phase.

Table 2-1 Phase and signal level interpretation

AXI4 channel	ARM::AXI4:: Phase::	Direction	Replied with (for VALIDs)	Signal-level interpretation
AR	AR_VALID	fw	AR_READY	**ARVALID** rise
AR	AR_READY	bw	-	**ARVALID** & **ARREADY**
R	R_VALID	bw	R_READY	**RVALID** rise & **!RLAST**
R	R_VALID_LAST	bw	R_READY	**RVALID** rise & **RLAST**
R	R_READY	fw	-	**RVALID** & **RREADY**
AW	AW_VALID	fw	AW_READY	**AWVALID** rise
AW	AW_READY	bw	-	**AWVALID** & **AWREADY**
W	W_VALID	fw	W_READY	**WVALID** rise & **!WLAST**
W	W_VALID_LAST	fw	W_READY	**WVALID** rise & **WLAST**
W	W_READY	bw	-	**WVALID** & **WREADY**

Table 2-1 Phase and signal level interpretation (continued)

AXI4 channel	ARM::AXI4:: Phase::	Direction	Replied with (for VALIDs)	Signal-level interpretation
B	B_VALID	bw	B_READY	**BVALID** rise
B	B_READY	fw	-	**BVALID** & **BREADY**
AC	AC_VALID	bw	AC_READY	**ACVALID** rise
AC	AC_READY	fw	-	**ACVALID** & **ACREADY**
CR	CR_VALID	fw	CR_READY	**CRVALID** rise
CR	CR_READY	bw	-	**CRVALID** & **CRREADY**
CD	CD_VALID	fw	CD_READY	**CDVALID** rise & **!CDLAST**
CD	CD_VALID_LAST	fw	CD_READY	**CDVALID** rise & **CDLAST**
CD	CD_READY	bw	-	**CDVALID** & **CDREADY**
RACK	RACK	fw	-	**RACK** rise
WACK	WACK	fw	-	**WACK** rise

The AXI4 data-passing channels R, W, and CD, have an extra VALID_LAST phase which indicates that a data beat is the last one of that burst. This VALID_LAST phase must be used with the last beat of a burst on those channels for all burst length. The matching READY phase of VALID_LAST is the standard READY phase for that AXI channel.

Note

There are no READY_LAST phases.

Last beats are marked with different phases to make tracking transition progress easier for ports where there is no requirement to count burst data beats.

2.3 ACE WACK and RACK signals

On ACE ports, **WACK** and **RACK** signals indicate that a master has completed a transaction and can be snooped concerning that data. In this library that signal is communicated as a forward call with the phase set to **RACK** or **WACK**. The payload communicated with that call must be the correct payload for the transaction being acknowledged. The receiving port must reply with TLM_ACCEPTED. The timing requirements of **WACK** and **RACK** are similar to other AXI channels VALID and READY state machine, but without a REQ state.

Separate each call of either **WACK** or **RACK** with a posedge of the clock signal to allow the channel to return to the CLEAR state.

Chapter 3
AXI4 payload

This chapter describes the single AXI4 payload type that is used by the AXI4 protocol. It is designed for use by a range of abstractions from purely behavioral untimed models, to interfacing with RTL signal level simulations at a signal and cycle-accurate level. It contains the following sections:

3.1 Payload generation

Payloads are reference-counted that are managed by a single global payload pool. Local payloads that are allocated on the stack are not permitted and are protected against by the payload having a private constructor.

A new AXI4 payload is constructed by calling ARM::AXI4::Payload::new_payload() with arguments which are essential for the construction and interpretation of data of the payload. These arguments which are explained in *Payload basic fields* on page 3-3 are:

- command
- address
- size
- len
- burst

Except for the fields that are passed as arguments to the constructor, and unless otherwise stated, newly generated payloads have all other fields set to 0.

The payload is allocated with a reference count of 1. You can increment and decrement the reference count of the payload by using the ref() and unref() functions respectively. It is the duty of the model that called the constructor to unref() the payload when it is no longer holding any pointers to it. All models which keep a pointer to a payload must ref() the payload and unref() it when the pointer is discarded.

3.2 Payload basic fields

The payload can represent any AXI4 transaction. The fields that it contains use the same enumeration encodings as the hardware implementation as listed in the *Arm® AMBA® AXI Protocol Specification, AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*. These fields are:

3.2.1 Command

This field consists of the following:

Name: command

Type: ARM::AXI4::Command

Getter: Command get_command() const

Settler: N/A (fixed at payload construction)

RTL signals: N/A (not communicated as a signal)

Default value: N/A (set by the constructor)

The Command field indicates on which AXI address channel the transaction was initiated, that is, AR, AW or AC. It is set during payload construction and cannot be altered after construction. You can also derive this information from the AXI channel which corresponds to the phase accompanying the payload. But it is also carried in the payload as a convenience.

3.2.2 Len

This field consists of the following:

Name: len

Type: uint8_t

Getter: uint8_t get_len() const

Settler: N/A (fixed at payload construction)

RTL signals: **ARLEN**, **AWLEN**

Default value: N/A (set by the constructor)

The Len field indicates the burst length of the transaction. It is set during payload construction and cannot be altered after construction. The beat count for a transaction is equal to len + 1. This encoding matches that of **ARLEN** and **AWLEN** signals. Although, in RTL, the number of data beats of a snoop transaction is implicitly defined by the cache line length and data channel width, when constructing an Arm TLM snoop payload, the Len field must be set to match the number of data beats the transaction could potentially return, that is, set to number_of_beats – 1.

3.2.3 Size

This field consists of the following:

Name: size

Type: ARM::AXI4::Size

Getter: Size get_size() const

Settler: N/A (fixed at payload construction)

RTL signals: **ARSIZE**, **AWSIZE**

Default value: N/A (set by the constructor)

The Size field indicates the size of each data beat. It is set during payload construction and cannot be altered after construction. The enumeration encodes the range of allowed beat size that is, 1 to 128, and matches that used by the **ARSIZE** and **AWSIZE** signals. The enumeration uses the beat size in bytes. For example, 128-bit wide transactions must set size to SIZE_16. The beat element size in bytes can be calculated using 1 << size. In snoop operations, this field must be set to the width of the data channel the transaction is operating across.

3.2.4 Burst

This field consists of the following:

Name: burst

Type: ARM::AXI4::Burst

Getter: Burst get_burst() const

Settler: N/A (fixed at payload construction)

RTL signals: **ARBURST**, **AWBURST**

Default value: N/A (set by the constructor, defaults to INCR if not specified)

The Burst field indicates the burst type of the transaction. It is set during payload construction and cannot be altered after construction. The enumeration matches that used by the **ARBURST** and **AWBURST** signals. The burst field is optional in the new_payload constructor and defaults INCR. The field is only relevant in read and writes operations. In snoop operations the field must be ignored and set to INCR.

3.2.5 Address

This field consists of the following:

Name: address

Type: uint64_t

Getter: uint64_t get_address() const

Settler: void set_address(uint64_t new_address)

RTL signals: **ARDDR**, **AWDDR**, **ACADDR**

Default value: N/A (set by the constructor)

The Address field indicates the address of the transaction. It is set during payload construction and can be altered after construction unless the change alters the interpretation of a wrapping burst beat orderings. The field matches that used by the **ARADDR**, **AWADDR**, and **ACADDR** signals.

In fixed and incrementing burst transactions, the address refers to the base address of the memory accessed. In wrapping bursts, the address points to the critical beat of the transaction. For functional memory models, it is the base address that is relevant to the operation and this is extractable by using the get_base_address() function. In wrapping burst transactions parts of the address determine the data beat ordering and so setting the address must only be performed if it does not affect these bits. To test whether an address is valid, mask both new and old addresses with payload->get_len() << payload->get_size() and compare them.

3.2.6 ID

This field consists of the following:

Name: id

Type: uint32_t

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARID**, **RID**, **AWID, BID**

Default value: 0

The ID field indicates the ID of the transaction. It is set to 0 at payload construction and can be altered after construction. The field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.7 Lock

This field consists of the following:

Name: lock

Type: ARM::AXI4::Lock

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARLOCK**, **AWLOCK**

Default value: LOCK_NORMAL (0)

The Lock field indicates the exclusiveness of the transaction. The enumeration matches that used by the **ARLOCK** and **AWLOCK** signals. It is set to LOCK_NORMAL at payload construction and can be altered after construction. The field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.8 Cache

This field consists of the following:

Name: cache

Type: ARM::AXI4::Cache

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARCACHE**, **AWCACHE**

Default value: CACHE_AR_DEVICE_NB or CACHE_AW_DEVICE_NB (0)

The Cache field indicates the cacheability of the transaction. The enumeration matches that used by the **ARCACHE** and **AWCACHE** signals. It is set to CACHE_AR_DEVICE_NB or CACHE_AW_DEVICE_NB at payload construction and can be altered after construction.

The element is accessible as:

- An integer value.
- An enumeration for read and write transactions.
- A set of bitfields using the CacheBitEnum type allowing access to the individual B, M, A, and AO bits.

The field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.9 Prot

This field consists of the following:

Name: prot

Type: ARM::AXI4::Prot

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARPOT**, **AWPROT**, **ACPROT**

Default value: PROT_D_S_UP (0)

The Prot field indicates the protection level of the transaction. The enumeration matches that used by the **ARPROT**, **AWPROT**, and **ACPROT** signals. It is set to PROT_D_S_UP (data, secure, unprivileged) at payload construction and can be altered after construction. The element is accessible as an integer value, an enumeration, and as a set of bitfields using the ProtBitEnum type allowing access to the individual P, NS and I bits.

3.2.10 QoS

This field consists of the following:

Name: qos

Type: uint8_t

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARQOS**, **AWQOS**

Default value: 0

The QoS field indicates the quality-of-service value of the transaction. It is set to 0 at payload construction and can be altered after construction. The field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.11 Region

This field consists of the following:

Name: region

Type: uint8_t

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARREGION**, **AWREGION**

Default value: 0

The Region field indicates the region of the transaction. It is set to 0 at payload construction and can be altered after construction. The field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.12 User

This field consists of the following:

Name: user

Type: uint64_t

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARUSER**, **RUSER**, **AWUSER**, **WUSER**, **BUSER**

Default value: 0

The User field indicates the user-defined signaling of the transaction. It is set to 0 at payload construction and can be altered after. The field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.13 Snoop

This field consists of the following:

Name: snoop

Type: ARM::AXI4::Snoop

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARSNOOP**, **AWSNOOP**, **ACSNOOP**

Default value: SNOOP_AW_WRITE_NO_SNOOP or SNOOP_AR_READ_NO_SNOOP or SNOOP_AC_READ_ONCE (0)

The Snoop field indicates the transaction coherency type. The enumeration matches that used by the **ARSNOOP**, **AWSNOOP**, and **ACSNOOP** signals. It is set to NO_SNOOP at payload construction and can be altered after construction.

3.2.14 Domain

This field consists of the following:

Name: domain

Type: ARM::AXI4::Domain

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARDOMAIN**, **AWDOMAIN**

Default value: DOMAIN_NSH(0)

The Domain field indicates the shareability domain of the transaction. The enumeration matches that used by the **ARDOMAIN** and **AWDOMAIN** signals. It is set to DOMAIN_NSH, which is not shared at payload construction and can be altered after construction. This field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.15 Bar

This field consists of the following:

Name: bar

Type: ARM::AXI4::Bar

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARBAR**, **AWBAR**

Default value: BAR_NORM (0)

The Bar field indicates whether the operation is a barrier transaction. The enumeration matches that used by the **ARBAR** and **AWBAR** signals. It is set to BAR_NORM, which is normal access, at payload construction and can be altered after construction. This field is only relevant in read and writes operations. In snoop operations, this field must be ignored.

3.2.16 Unique

This field consists of the following:

Name: unique

Type: bool

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **AWUNIQUE**

Default value: false

The Unique field is used with various write transactions to improve the operation of lower levels of the cache hierarchy. It is set to false at payload construction and can be altered after construction. This field is only relevant in writes operations. In read and snoop operations, this field must be ignored.

3.2.17 Atop

This field consists of the following:

Name: atop

Type: ARM::AXI4::Atop

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **AWATOP**

Default value: false

The Atop field is used by atomic operations. It is set to ATOP_NON_ATOMIC at payload construction and can be altered after construction. This field is only relevant in writes operations. In read and snoop operations this field must be ignored.

3.2.18 VMID Extension

This field consists of the following:

Name: vmid_ext

Type: uint8_t

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **ARVMIDEXT**, **ACVMIDEXT**

Default value: 0

The *Virtual Memory ID* (VMID) extension field indicates the high-order bits of the VMID used in *Distributed Virtual Memory* (DVM) transactions. It is set to 0 at payload construction and can be altered after. The field is only relevant in DVM carrying read and snoop operations. In write operations this field must be ignored.

3.2.19 Stash NID

This field consists of the following:

Name: stash_nid

Type: uint16_t

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **AWSTASHNID**

Default value: 0

The Stash NID field indicates the node identifier of the physical interface that is the target interface for the cache stash operation. It is set to 0 at payload construction and can be altered after. The field is only relevant in write operations. In read and snoop operations, this field must be ignored.

3.2.20 Stash NID Valid

This field consists of the following:

Name: stash_nid_valid

Type: bool

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **AWSTASHNIDEN**

Default value: false

The Stash NID Valid field indicates that the Stash NID field is valid and must be used. It is set to false at payload construction and can be altered after. The field is only relevant in write operations. In read and snoop operations, this field must be ignored.

3.2.21 Stash LPID

This field consists of the following:

Name: stash_lpid

Type: uint8_t

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **AWSTASHLPID**

Default value: 0

The Stash LPID field indicates the logical processor subunit that is associated with the physical interface that is the target for the cache stash operation. It is set to 0 at payload construction and can be altered after. The field is only relevant in write operations. In read and snoop operations, this field must be ignored.

3.2.22 Stash LPID Valid

This field consists of the following:

Name: stash_lpid_valid

Type: bool

Getter: N/A (directly accessible)

Settler: N/A (directly accessible)

RTL signals: **AWSTASHLPIDEN**

Default value: false

The Stash LPID Valid field indicates that the Stash LPID field is valid and must be used. It is set to false at payload construction and can be altered after. The field is only relevant in write operations. In read and snoop operations, this field must be ignored.

3.2.23 Resp

This field consists of the following:

Name: resp

Type: ARM::AXI4::Resp

Getter: Resp get_resp() const

Settler: void set_resp(Resp resp)

RTL signals: **RRESP**, **BRESP**, **CRRESP**

Default value: RESP_OKAY (0)

The Resp field indicates the response status of the transaction. It is set to RESP_OKAY at payload construction and can be altered after construction. The enumeration matches that used by the **RRESP**, **BRESP**, and **CRRESP** signals with the addition of the RESP_INCONSISTENT value.

In read operations, each beat can have a different resp value. If resp values for all beats are equal, resp for the whole payload is set to that value. If beat resps are not all equal, the resp field is set to RESP_INCONSISTENT and the resp values for each beat are accessible with the read_out_resps() function that is described in *Read* on page 3-13.

Note

Do not set this field to RESP_INCONSISTENT directly.

3.3 Helper functions

A number of extra helper functions are provided to aid in working with AXI4 payloads. These functions are:

- get_beat_count()
- get_beat_data_length ()
- get_beats_complete ()
- get_data_length()
- get_base_address()

Function: get_beat_count()

Prototype: unsigned get_beat_count() const

Because the value of the len field within the payload can be counter-intuitive, get_beat_count() provides the correct number of data beats for the payload.

Function: get_beat_data_length ()

Prototype: std::size_t get_beat_data_length () const

The get_beat_data_length () function provides the size (in bytes) of each beat in the payload. Before calling the per beat data copy out functions, a buffer must be allocated of at least the size that is returned by get_beat_data_length().

Function: get_beats_complete ()

Prototype: unsigned get_beats_complete () const

The get_beats_complete () function provides the number of beats of data that has been entered into the payload. Before calling any data copy out functions, get_beats_complete () can be used to indicate whether copying the data beat from the payload would result in an error due to the data not yet having been entered into the payload.

Function: get_data_length()

Prototype: std::size_t get_data_length() const

The get_data_length() function provides the size (in bytes) of the data payload of the whole transaction. Before calling the data copy out functions, a buffer must be allocated of at least the size that is returned by get_data_length().

Function: get_base_address()

Prototype: uint64_t get_base_address() const

In wrapping transactions, the address field specifies an address of the critical beat rather than the base of the transaction. The get_base_address() function provides the base address of any transaction performing the appropriate rounding down for wrapping bursts.

3.4 Data and responses

Unlike the Generic TLM 2.0 payload, in ARM::AXI4::Payload the data that is carried for the transaction is not directly accessible. Data is accessed through copy in and copy out access functions. Copying out data is only permitted on data that has previously been copied in to the payload.

If the transaction is communicated across a socket as beats, copying out the data of the entire transaction is only permitted when the last data beat has been received. If all data is entered in one function call, the payload must have this data set before communicating the first data beat of the transaction.

These functions are:

- *Read*
- *Write*
- *Snoop* on page 3-14

3.4.1 Read

These functions consists of the following:

Function: read_in

Prototype: void read_in(const uint8_t* data, Resp* resp = NULL)

Copy in the data for a whole read transaction from an array of get_data_length() bytes. resp is an optional array of per-beat responses get_beat_count() Resps long. If not provided, the resp value defaults to RESP_OKAY.

Function: read_out

Prototype: void read_out(uint8_t* data) const

Copy out the data for a whole read transaction to an array of get_data_length() bytes.

Function: read_out_resps

Prototype: void read_out_resps(Resp* resp) const

Copy out the beat response values for a whole read transaction to an array get_beat_count() Resps long.

3.4.2 Write

These functions consists of the following:

Function: write_in

Prototype: void write_in(const uint8_t* data, const uint8_t* strobe = NULL)

Copy in the data for a whole write transaction from an array get_data_length() bytes long. An optional array of byte strobes ceil(get_data_length() / 8.0) bytes long can be passed to select which bytes to write. The strobes are organized as one bit per data byte with the lowest bit of byte strobe[0] corresponding to the lowest byte of data. If the strobe is NULL that is, the default value if not specified, then all bytes are written.

Function: write_out

Prototype: void write_out(uint8_t* data) const

Copy out the data for a whole write transaction into an array get_data_length() bytes long. The data strobes are respected and the original values are preserved in the destination for bytes where the strobe is disabled.

Function: write_out_strobes

Prototype: void write_out_strobes(uint8_t* strobe) const

Copy out the strobes for a whole write transaction into an array ceil(get_data_length() / 8.0) bytes long.

3.4.3 Snoop

These functions consists of the following:

Function: snoop_in

Prototype: void snoop_in(const uint8_t* data)

Copy in the data for a whole snoop transaction from an array get_data_length() bytes long.

Function: snoop_out

Prototype: void snoop_out(uint8_t* data) const

Copy out the data for a whole snoop transaction into an array get_data_length() bytes long.

3.5 Beat level data processing

If the transaction is communicated across a socket as beats, it is permissible to enter the data for individual beats before that beat is communicated across the socket. The data of an individual beat can be copied out of the payload when that beat has communicated across the socket. The beat index that is supplied to the copy out access functions is the index of the beat in communication sequence, and not the address sequence.

These functions are:

- *Read*.
- *Write*.
- *Snoop* on page 3-16.

3.5.1 Read

These functions consists of the following:

Function: read_in_beat

Prototype: void read_in_beat(const uint8_t* data, Resp resp = RESP_OKAY)

Copy in the data for one beat of a read transaction from an array get_beat_data_length() bytes long. The beat response is set to resp. If not provided, resp defaults to RESP_OKAY.

Function: read_out_beat

Prototype: void read_out_beat(unsigned beat_index, uint8_t* data) const

Copy out the data for one beat of a read transaction into an array get_beat_data_length() bytes long. The first beat of a transaction has index 0. Only beats that have already been written into a payload can be read out.

Function: read_out_beat_resps

Prototype: Resp read_out_beat_resp(unsigned beat_index) const

Get the response for one beat of a read transaction. The first beat of a transaction has index 0. Only beats that have already been written into a payload can be copied out.

3.5.2 Write

These functions consists of the following:

Function: write_in_beat

Prototype: void write_in_beat(const uint8_t* data, const uint8_t* strobe = NULL)

Copy in the data for one beat of a write transaction. The write strobe is passed as an array ceil(get_beat_data_length() / 8.0) bytes long with the same strobe organization as for write_in() but with the lowest strobe bit corresponding to the beat data[0] rather than the whole data of the transaction.

Function: write_in_beat

Prototype: void write_in_beat(const uint8_t* data, uint64_t strobe)

Copy in the data for one beat of a write transaction where the beat is shorter than 64 bytes long (size <= SIZE_64). strobe encodes the byte strobes for the beat with the lowest bit of strobe being the strobe for data[0].

Function: write_out_beat

Prototype: void write_out_beat(unsigned beat_index, uint8_t* data) const

Copy out the data for one beat of a write transaction to an array get_beat_data_length() bytes long. The first beat of a transaction has index 0. Only beats that have already been written into a payload can be read out.

Function: write_out_beat_strobe

Prototype: void write_out_beat_strobe(unsigned beat_index, uint8_t* strobe) const

Copy out the strobes for one beat of a write transaction into an array ceil(get_beat_data_length() / 8.0) bytes long. The first beat of a transaction has index 0. Only beats that have already been written into a payload can have their strobe copied out. The strobes are organized as one bit per data byte with the lowest bit of byte strobe[0] corresponding to the lowest byte of data.

Function: write_out_beat_strobe

Prototype: uint64_t write_out_beat_strobe(unsigned beat_index) const

Get the strobe for one beat of a write transaction where the beat is shorter than 64 bytes long (get_size() <= SIZE_64). The returned strobe is organized with the lowest bit corresponding to the lowest address byte in the beat.

3.5.3 Snoop

These functions consists of the following:

Function: snoop_in_beat

Prototype: void snoop_in_beat(const uint8_t* data)

Copy in the data for one beat of a snoop transaction from an array get_beat_data_length() bytes long.

Function: snoop_out_beat

Prototype: void snoop_out_beat(unsigned beat_index, uint8_t* data) const

Copy out the data for one beat of a snoop transaction into an array get_beat_data_length() bytes long. The first beat of a transaction has index 0. Only beats that have already been written into a payload can have their strobe copied out.

3.6 Signal level support

To help with communication between signal level and TLM 2.0 models, a set of functions is provided to propagate data between the payload and raw signal level interfaces.

These functions are:

- *Read*.
- *Write*.
- *Snoop* on page 3-18.

3.6.1 Read

These functions consists of the following:

Function: read_in_beat_raw

Prototype: void read_in_beat_raw(Size width, const uint8_t* data, Resp resp = RESP_OKAY)

Copy in the data for one beat of a read transaction that is supplied in raw signal level format. width is the width of the data channel (in bytes) in the signal level implementation of the AXI4 port. data is an array (1 << width) bytes long.

Function: read_out_beat_raw

Prototype: void read_out_beat_raw(Size width, unsigned beat_index, uint8_t* data) const

Copy out the data for one beat of a read transaction that is supplied in raw signal level format. width is the width of the data channel (in bytes) in the signal level implementation of the AXI4 port. data is an array (1 << width) bytes long. The first beat of a transaction has index 0. Only beats that have already been written into a payload can be read out.

3.6.2 Write

These functions consists of the following:

Function: write_in_beat_raw

Prototype: void write_in_beat_raw(Size width, const uint8_t* data, const uint8_t* strobe)

Copy in the data for one beat of a write transaction that is supplied in raw signal level format. width is the width of the data channel (in bytes) in the signal level implementation of the AXI4 port. data is an array (1 << width) bytes long. The write strobe is passed as an array ceil((1 << width) / 8.0) bytes long with the same strobe organization as for write_in() but with the lowest strobe bit corresponding to the signal level beat data[0] rather than the whole data of the transaction.

Function: write_in_beat_raw

Prototype: void write_in_beat_raw(Size width, const uint8_t* data, uint64_t strobe)

Copy in the data for one beat of a write transaction that is supplied in raw signal level format where width <= SIZE_64. width is the width of the data channel (in bytes) in the signal level implementation of the AXI4 port. data is an array (1 << width) bytes long. Strobes are organized with the lowest bit being the strobe for byte data[0].

Function: write_out_beat_raw

Prototype: void write_out_beat_raw(Size width, unsigned beat_index, uint8_t* data) const

Copy out the data for one beat of a write transaction that is supplied in raw signal level format. data is an array width bytes long. The first beat of a transaction has index 0. Only beats that have already been written into a payload can have their strobe copied out.

Function: write_out_beat_raw_strobe

Prototype: void write_out_beat_raw_strobe(Size width, unsigned beat_index, uint8_t* strobe) const

Get the strobe for one beat of a write transaction. width is the width of the data channel (in bytes) in the signal level implementation of the AXI4 port. The write strobe is passed as an array ceil(width / 8.0) bytes long with the same strobe organization as for write_in() but with the lowest strobe bit corresponding to the lowest data byte of the signal level beat rather than the whole data of the transaction.

Function: write_out_beat_raw_strobe

Prototype: uint64_t write_out_beat_raw_strobe(Size width, unsigned beat_index) const

Get the strobe for one beat of a write transaction that is supplied in raw signal level format where width <= SIZE_64. width is the width of the data channel (in bytes) in the signal level implementation of the AXI4 port. Strobes are organized with the lowest bit being the strobe for byte data[0]. The first beat of a transaction has index 0. Only beats that have already been written into a payload can have their strobe copied out.

3.6.3 Snoop

Interfacing to the signal level CD AXI channels can be implemented using signal level read functions.

3.7 Payload propagation

In a simple system, payloads are generated at the transaction initiator, they are transmitted to the target model which fills in the response fields and passes the payload back to the initiator. In more complex systems, the payload can pass through several models between the initiator and the target. It is legal and encouraged for a module, which only propagates the transaction unchanged, to forward the payload that was received.

Other models must amend the payload before propagating it. A model which receives a transaction over a socket is only permitted to write the data and response fields and only if it is the final target model. These models should not change other fields because the payload can still be in use by other models.

Two methods are provided to enable models to create payloads that are derived from payloads that have been received.

3.7.1 Descend

The descend() function creates a payload and it copies all fields from a previous payload to create a copy of that payload. The function takes the same parameters as new_payload() and there is no restriction on what kind of payload is generated from the parent. All fields that are not set by the constructor can be altered before propagating just like a new payload. Descend can be called multiple times on a single payload creating multiple derived transactions.

Arm recommends you to use descend when creating derived payloads rather than directly creating payloads. See *UID* on page 3-20 and *Parent* on page 3-20 for features which can help when handling derived payloads. When responses of descended transactions return, it is the job of the model which descended that payload to perform some tasks. These tasks are to copy the response and data from the descended payload to the original before propagating the original back upstream.

3.7.2 Clone

Models such as simple interconnects, address mappers, and other models which change transaction request fields but do not alter the command or the burst qualifications may use the clone() function to create a payload. The function creates a payload that is based on the original, similar to descend(). However, data and response fields of the payload are shared between the cloned payload and the original. The new payload has all fields set to the same values as the original. The following construction time fields cannot be changed:

- command
- size
- len
- burst
- bits of the address

All other fields can be changed before the payload is propagated.

Because the response and data fields are shared, a target model which sets the response and data values of the cloned payload also sets those of the original. For this reason, each payload can be cloned at most once. There is no requirement for the model which cloned the payload to copy the transaction response between the cloned and original.

3.8 Advanced payload fields

A number of fields are attached to all payloads to help the programmer working with ARM::AXI4::Payload. These fields have no representation in the signal level implementation and are not expected to have any influence on the behavior of a model.

These fields are:

3.8.1 UID

This field consists of the following:

Name: uid

Type: uint64_t const

Getter: N/A (directly accessible)

Settler: N/A (const)

RTL signals: N/A (simulation construct)

Default value: New unique value to each constructed payload

The UID field is a unique payload ID that you can use to reference a specific payload. It is set to a unique value at payload construction and cannot be altered after construction. The uid is unique for each payload that is constructed and cannot be reused even if a payload is destroyed. The uid is not copied by clone or descend operations and child payloads have a new, unique uid value. The uid of 0 is never allocated to a payload. It can be used as a sentinel value where uids are used as key values outside ARM::AXI4::Payload.

3.8.2 Parent

This field consists of the following:

Name: parent

Type: Payload* const

Getter: N/A (directly accessible)

Settler: N/A (const)

RTL signals: N/A (simulation construct)

Default value: NULL (unless descended or cloned)

The Parent field is a pointer to the payload that this payload is descended from. In newly constructed payloads, it is set to NULL unless the payload is constructed by descending or cloning. If this is the case, the parent field is set to point to the original payload. It cannot be altered post payload construction.

Descended and cloned payloads that reference a parent, increment the reference count of that parent, and decrement the parent reference count when they are destroyed. It is always safe to look at the parent of any payload, if it has one, and no manual referencing is required. It is possible to walk up the parent tree to find the original transaction that triggered the current one.

3.9 Extension system

The payload extension system allows extra fields to be added to payloads. All payloads have space that is allocated to hold all extensions whether they are set of not.

Values of extensions are copied when a payload is cloned or descended. Extensions are not shared between payloads, even if a transaction is cloned. Changing the value in the clone does not change the value in the original payload. Extension fields in new payloads are cleared to 0. Each extension can be of any object type. If the extension type is a class, the constructor for that class is called upon payload construction, and the copy contractor is called when payloads are descended or cloned.

Note

- Extensions must be declared before any payload object is created.
- Because extensions are typically declared within models, no payloads are created by models before all models in a system are instantiated.

A payload extension handle is a class that provides access to a payload extension value. Constructing the handle during model instantiation adds the extension to the set allocated in payloads, if it has not already been added. The first handle for an extension must be created to register the extension before any payloads are created. Constructing two handles with the same name and type provides accessors to the same extension.

The following example code constructs an extension named CORE_ID of type uint32_t:

```
ARM::AXI4::PayloadExtension<uint32_t> core_id_accessor("CORE_ID");
```

Providing a payload to the accessor through the get() function returns a non-const reference to the value for that payload extension. The following example code demonstrates changing an extension value:

```
uint32_t& core_id_value = core_id_accessor.get(&payload);

core_id_value &= 0xF;
```