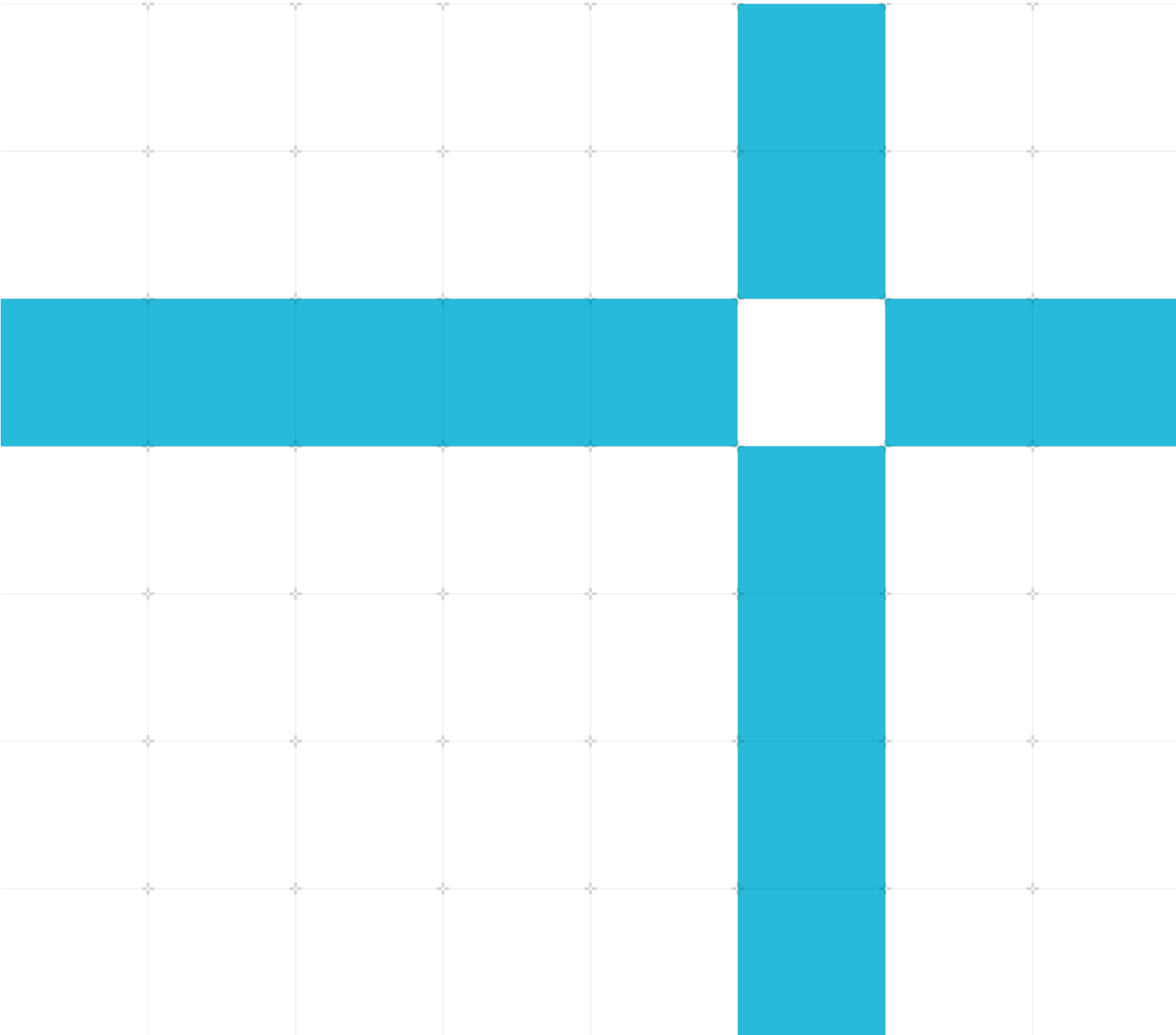
AMBA® TLM 2.0 Library

Reference Manual

Non-Confidential

Issue 02
101459_02_en



AMBA® TLM 2.0 Library

Reference Manual

Release information

Document history

Issue	Date	Confidentiality	Change
01 (1000-00)	1 January 2019	Non-Confidential	First release
02	1 June 2023	Non-Confidential	Second release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
(LES-PRE-20349)

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.

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on [Product Name], create a ticket on **https://support.developer.arm.com**.

To provide feedback on the document, fill the following survey: **https://developer.arm.com/documentation-feedback-survey.**

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email **terms@arm.com**.

Contents

1. Introduction

This document describes a C++ library of TLM 2.0-compatible type definitions for modeling AMBA® AXI, ACE, and CHI ports on SystemC models with approximate and cycle-accurate timing requirements.

1.1. Intended audience

The reader of this document is assumed to understand SystemC, TLM 2.0 Generic Protocol payloads and phases, and TLM 2.0 sockets. An understanding of the blocking and non-blocking transports used by TLM 2.0 and the motivations for the form of those mechanisms is also assumed.

1.2. Conventions

The following subsections describe conventions used in Arm documents.

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 the Arm Glossary for more information: **https://developer.arm.com/glossary**.

Typographical conventions

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` **`bold`**	Language keywords when used outside example code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses 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	Terms that have specific technical meanings as defined in the Arm® Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Convention	Use
	Recommendations. Not following these recommendations might lead to system failure or damage.
	Requirements for the system. Not following these requirements might result in system failure or damage.
	Requirements for the system. Not following these requirements will result in system failure or damage.
	An important piece of information that needs your attention.
	A useful tip that might make it easier, better, or faster to perform a task.
	A reminder of something important that relates to the information you are reading.

1.3. Useful resources

This document contains information that is specific to this product. See the following resources for other relevant information.

- Arm Non-Confidential documents are available on developer.arm.com/documentation. Each document link in the following tables provides direct access to the online version of the document.
- Arm Confidential documents are available to licensees only through the product package.

Arm architecture and specifications	Document ID	Confidentiality
AMBA® 5 CHI Architecture Specification	IHI 0050E.c	Non-Confidential
AMBA® AXI and ACE Protocol Specification	IHI 0022H	Non-Confidential

Non-Arm resources	Document ID	Organization
For information about the SystemC and TLM 2.0 standards, see *IEEE Standard for Standard SystemC Language Reference Manual, January 2012.*	IEEE Std 1666-2011	IEEE, https://www.ieee.org/

2. AMBA TLM 2.0 Library overview

The AMBA® Transaction-Level Modeling (TLM) 2.0 Library allows you to model and simulate Approximately-Timed (AT) and Cycle-Accurate (CA) AXI, ACE and CHI transactions and ports.

The supported versions of the AMBA AXI, ACE, and CHI protocols are:

- AMBA® AXI and ACE Protocol Specification issue H, and backward compatible with issue G, issue F, and issue E
- AMBA® 5 CHI Architecture Specification issue E, and backward compatible with issue D, issue C, and issue B

The AMBA TLM Library package includes:

- This document
- A pre-compiled binary library
- Examples of the library in use
- C++ source files

2.1. Header file structure

The AMBA TLM 2.0 Library is provided as a pre-compiled binary library and a collection of C++ header files. These header files define payload types, communication phases, base socket definitions, and types to identify the protocol used at particular sockets.

The following table summarizes the provided header files.

Table 2-1: C++ header files

Header file	Contents
`arm_axi4.h`	`#includes` all the header files needed to model AMBA AXI.
`arm_axi4_payload.h`	The constituent types required to describe the data and control payload of an AXI transaction. The payload types are described in 3 AXI protocol.
`arm_axi4_phase.h`	The phase type used to describe events in AXI for Approximately-Timed (AT) and Cycle-Accurate (CA) modeling. These phases are described in the context of the protocols that they form in 3.1 AXI phases.
`arm_axi4_socket.h`	Base socket definitions for initiator and target TLM 2.0 sockets for AXI ports. These definitions are described in 2.2 Socket types.
`arm_chi.h`	`#includes` all the header files needed to model AMBA CHI.
`arm_chi_payload.h`	The constituent types required to describe the data and control payload of a CHI transaction. The payload types are described in 4 CHI protocol.
`arm_chi_phase.h`	The phase type used to describe events in CHI Approximately-Timed (AT) and Cycle-Accurate (CA) modeling. These are described in the context of the protocols they form in 4.1 CHI phases.

Header file	Contents
`arm_chi_socket.h`	Base socket definitions for initiator and target TLM 2.0 sockets for CHI ports. These are described in 2.2 Socket types.
`arm_tlm_helpers.h`	Miscellaneous helper type definitions used by other header files.
`arm_tlm_protocol.h`	An enumeration of the varieties of AXI and CHI supportable by a socket. This enumeration is described in 2.2.1 Socket protocol and port width.
`arm_tlm_socket.h`	Contains common base socket definitions that contain protocol and port width definitions for a port but are not committed to a particular payload or protocol. These definitions are described in 2.2 Socket types and 2.2.1 Socket protocol and port width.

2.2. Socket types

Base initiator and target sockets are provided. A single socket models an entire AXI or CHI port including address, data, response, snoop, and ACE WACK/RACK channels as applicable. Ports on models should be derived from these base sockets.

Base sockets are provided as these types:

- `ARM::TLM::BaseTargetSocket`, derived from `tlm::tlm_target_socket`
- `ARM::TLM::BaseInitiatorSocket`, derived from `tlm::tlm_initiator_socket`

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

The following table lists the supported specializations.

Table 2-2: Supported specializations

Socket types	Applicability
`ARM::TLM::BaseTargetSocket<ARM::CHI::ProtocolType>` `ARM::TLM::BaseInitiatorSocket<ARM::CHI::ProtocolType>`	For CHI models
`ARM::TLM::BaseTargetSocket <ARM::AXI::ProtocolType>` `ARM::TLM::BaseInitiatorSocket <ARM::AXI::ProtocolType>`	For AXI models

The types `ARM::AXI::BaseTargetSocket` and `ARM::AXI::BaseInitiatorSocket` are provided with `ARM::AXI::ProtocolType` as their default specialization.

2.2.1. Socket protocol and port width

In addition to the port name, the two base socket types are passed on construction. There are two arguments that configure the expected communications on the port.

The following table describes the arguments.

Table 2-3: Socket protocol and port width arguments

Argument	Type	Meaning
`protocol`	`ARM::TLM::Protocol`	The AXI or CHI protocol variant expected on the port.
`port_width`	`unsigned`	The width of the data channels of the port in bits. This must be a power of two and be valid according to the AMBA® AXI and ACE Protocol Specification or the AMBA® 5 CHI Architecture Specification.

The following table describes the valid values for the socket protocol argument.

Table 2-4: Protocol values

Protocol value	Description
PROTOCOL_ACE	Full ACE with DVM, barriers, snoop channels and ACE RACK/WACK channels
PROTOCOL_ACE_LITE	ACE-Lite. ACE without snoop channels and RACK/WACK channels
PROTOCOL_ACE_LITE_DVM	ACE-Lite with additional DVM support
PROTOCOL_AXI4	AXI4 without any ACE features
PROTOCOL_AXI4_LITE	AXI4-Lite. AXI with simpler payload options
PROTOCOL_AXI5	AXI5 without any ACE features
PROTOCOL_AXI5_LITE	AXI5-Lite. AXI with simpler payload options
PROTOCOL_ACE5	ACE5. Full ACE5
PROTOCOL_ACE5_LITE	ACE5-Lite. ACE5 without snoop channels and RACK/WACK channels
PROTOCOL_ACE5_LITE_DVM	ACE-Lite with additional DVM support
PROTOCOL_ACE5_LITE_ACP	ACE-Lite with additional ACP support
PROTOCOL_CHI_B	CHI implementing the CHI.B standard
PROTOCOL_CHI_C	CHI implementing the CHI.C standard
PROTOCOL_CHI_D	CHI implementing the CHI.D standard
PROTOCOL_CHI_E	CHI implementing the CHI.E standard

2.2.2. Binding sockets

Sockets are bound together according to the standard TLM 2.0 method. `ARM::TLM::BaseTargetSocket` and `ARM::TLM::BaseInitiatorSocket` overload the member function `bind()` to ensure that the protocol and port width of bound sockets exactly match. Connecting sockets with different protocols or port widths result in a SystemC error.

2.2.3. Supported transport interfaces

The AXI and CHI protocols support only a subset of the interfaces that the TLM 2.0 types, `tlm::tlm_fw_transport_if` and `tlm::tlm_bw_transport_if` provide.

These transport call types are supported:

Non-blocking transport

Non-blocking forward and backward calls are supported by all sockets. All protocols that this library supports use these calls as the standard communication method.

Blocking transport

Blocking transport is not supported by the protocols that this library supports.

DMI

DMI is not supported by the protocols that this library supports

Debug transport

Sockets must implement the `transport_dbg()` call. Not all models can deal with debug transactions correctly, therefore, it is always valid for a model to return 0 and so decline to accept debug transactions.

Simple sockets

Sockets `ARM::TLM::SimpleInitiatorSocket` and `ARM::TLM::SimpleTargetSocket` are provided to implement non-blocking transport and debug transport using member functions on objects other than sockets. These work in a similar way to the TLM 2.0 types, `tlm_utils::tlm_simple_initiator_socket` and `tlm_utils::tlm_simple_target_socket`, but do not provide any blocking to non-blocking adaptation.

`ARM::AXI::SimpleInitiatorSocket` and `ARM::AXI::SimpleTargetSocket` are provided for use with AXI protocols.

2.3. 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 clock periods are supported:

Communicate

When TLM 2.0 interface calls are made between ports

Update

When internal state updates are made in response to communications

Two periods are required so that all communications in a cycle are known to occur before a state update at the start of the next cycle takes place. A model does not require a free-running clock signal but must respect the communicate/update clock periods of the model to which it is connected.

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

AMBA TLM requires at minimum that where a clock signal associated with a socket is present, communication does not take place that is triggered by the positive-going event of that clock. We recommend that all communication is triggered only by the negative-going edge of the clock.

For this reason, this document refers to events causing communication as negedge and those causing state update as posedge.

3. AXI protocol

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

The AXI protocol is designed to be used at a range of abstractions from purely behavioral untimed models to interfacing with RTL signal level simulations at a signal and cycle-accurate level.

3.1. AXI phases

AXI communications use several channels and most of these communications use READY and VALID phases.

AXI channels

Each AXI port logically consists of several AXI channels. The complete set of channels is:

- AW
- W
- B
- AR
- R
- AC
- CR
- CD
- WACK
- RACK
- WQOSACCEPT
- RQOSACCEPT

AXI channel phases

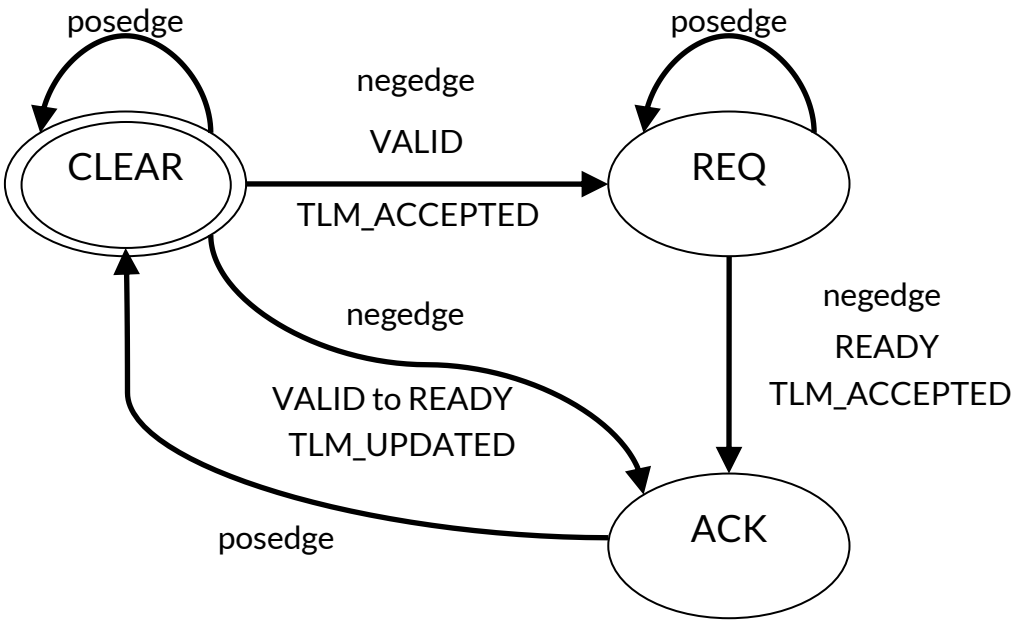
All the channels of a port are modeled with a single TLM 2.0 socket. Each channel has a pair of phase enumeration values from the type `ARM::AXI::Phase` representing VALID and READY signaling on that channel. For example, AXI channel AW has phases `AW_VALID` and `AW_READY`. Two channels do not have a pair of phase enumeration values, *ACK and *QOSACCEPT.

With non-blocking communication, these functions are used:

- AXI channels from Initiator port to Target port send their VALID event using `nb_transport_fw()`. They 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 Target port to Initiator port (for example, B) send VALID with `nb_transport_bw()` and READY with `nb_transport_fw()`.

Figure 3-1 AXI Channel State Machine



Each AXI channel, except *ACK and *QOSACCEPT, obeys the state machine shown in Figure 3-1. 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` receives the response `AW_READY`), which is analogous to the READY signal being observed after a raised VALID signal. READY can be communicated by either:

- Immediately changing the `nb_transport_fw/bw()` phase to READY, passed by reference, and returning with `tlm::TLM_UPDATED`.
- Responding later in the opposite direction with a READY phase.

These two options are shown in Figure 3-1 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.
- The response is in a different cycle (after the next posedge) it indicates a delayed READY.

Transition from the ACK state back to CLEAR in Figure 3-1 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 remain to ensure that the next update period in the agreed clocking regime has passed before the next VALID phase can be sent.

Although the AMBA® AXI and ACE Protocol Specification allows the READY signal to rise before the VALID signal has arrived, this library does not model this behavior. Therefore, a READY call must only be generated as a response to a received VALID phase. This means a VALID phase indicates that both READY and VALID signals are high.

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

The following table includes:

- All phases
- The direction phases should be transmitted, bw or fw, that is, backward or forward according to the `nb_transport_fw` or `nb_transport_bw` calls mentioned in 3.1 AXI phases.
- The response for VALID calls. Optionally, immediately using the `tlm::TLM_UPDATED` response.
- The signal-level interpretation for each phase

Table 3-1: AXI channel phases

AXI channel	ARM::AXI::Phase::	Direction	Response (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
B	B_VALID	bw	B_READY	BVALID rise
B	B_VALID_COMP	bw	B_READY	BVALID rise & BCOMP & !BPERSIST
B	B_VALID_PERSIST	bw	B_READY	BVALID rise & !BCOMP & BPERSIST

AXI channel	ARM::AXI::Phase::	Direction	Response (for VALIDs)	Signal-level interpretation
B	B_VALID_COMP_PERSIST	bw	B_READY	BVALID rise & BCOMP & BPERSIST
B	B_VALID_TAGMATCH	bw	B_READY	BVALID rise & !BCOMP & BTAGMATCH[1]
B	B_VALID_COMP_TAGMATCH	bw	B_READY	BVALID rise & BCOMP & BTAGMATCH[1]
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
RQOSACCEPT	RQOSACCEPT	bw	-	RQOSACCEPT change
WQOSACCEPT	WQOSACCEPT	bw	-	WQOSACCEPT change

AXI data passing channels (R, W and CD) have an additional VALID_LAST phase which indicates that a data beat is the last one of that burst. This VALID_LAST phase should be used with the last beat of a burst on those channels for all burst lengths. VALID_LAST's matching READY phase is the standard READY phase for that AXI channel. There are no READY_LAST phases. Last beats are marked with different phases to make tracking transaction progress easier for ports that you do not want to count burst data beats.

In B channels which support cache maintenance operations for persistence or Memory Tagging Extension (MTE), the B_VALID phases are extended to communicate the value of BCOMP, BPERSIST, and BTAGMATCH[1] signals. The value of TAGMATCH[0] (pass or fail) is available in the payload.

When receiving a phase, if you want to compare it to one of the values in the preceding table, the phase should be stripped to remove any additional fields in it. To strip the phase, use the `Phase phase_strip(Phase phase)` function.

3.1.1. ACE WACK and RACK signals

On ACE ports, WACK and RACK signals indicate that an Initiator has completed a transaction and can be snooped with respect to 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 should 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 the VALID/READY state machine of other AXI channels, but without a REQ state.

3.1.2. QOSACCEPT

On ports that support it, the values of RQOSACCEPT and WQOSACCEPT signals can be communicated across the TLM channel. These are communicated using bw calls.

The payload used in the call should not be accessed by the call recipient. The caller can either use a dummy payload, provided by the `AXI::Payload::get_dummy()` function, or any other payload. The value of the QOSACCEPT signal is communicated within the phase and can be extracted using `uint8_t phase_get_qos_accept(Phase phase)` function. It can be set using the `void phase_set_qos_accept(Phase &phase, uint8_t qos_accept)` function. Because the QoS value is encoded within the phase value, the phase must be stripped before it can be tested for which phase value it holds. This is done using the `Phase phase_strip(Phase phase)` function.

3.1.3. RCHUNKNUM and RCHUNKSTRB

In AXI chunking transactions, the RCHUNKNUM and RCHUNKSTRB signals are available in the phase of the R beat phases.

The phase_get_chunk_number and phase_set_chunk_number functions access the RCHUNKNUM value. The phase_get_chunk_strobe and phase_set_chunk_strobe access the RCHUNKSTRB value. As with other phases that encode additional information, the phase should be stripped before being compared to one of the enumeration values.

3.2. AXI payload generation

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

A new AXI payload is constructed by calling `ARM::AXI::Payload::new_payload()` with arguments that are essential for the construction and interpretation of the data of the payload. The command, address, size, len, and burst arguments are described in 3.3 AXI payload fields.

Except for the fields 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. The reference count of the payload can be incremented and decremented using the `ref()` and `unref()` functions respectively. It is the duty of the model that called the constructor to `unref()` the payload once it is no longer holding any pointers to it. All models that keep a pointer to a payload should `ref()` the payload and `unref()` it once the pointer is discarded.

3.3. AXI payload fields

The payload can represent any AXI transaction.

The fields the payload contains use the same enumeration encodings as the hardware implementation as listed in the AMBA® AXI and ACE Protocol Specification.

3.3.1. Command field

The `command` field indicates on what AXI address channel the transaction was initiated, AR, AW, or AC. It is set during payload construction and cannot be altered after construction.

This information can be derived from the AXI channel that corresponds to the phase accompanying the payload but is also carried in the payload.

Name	`command`
Type	`ARM::AXI::Command`
Getter	`Command get_command() const`
Setter	N/A, fixed at payload construction
RTL signals	N/A, not communicated as a signal
Default value	N/A, set by the constructor

3.3.2. Len field

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 the encoding of the ARLEN and AWLEN signals.

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 a 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`.

Name	`len`
Type	`uint8_t`

Getter	`uint8_t get_len() const`
Setter	N/A, fixed at payload construction
RTL signals	ARLEN, AWLEN
Default value	N/A, set by the constructor

3.3.3. Size field

The `size` field indicates the size of each data beat in bytes. It is set during payload construction and cannot be altered after construction.

The enumeration encodes the range of allowed beat sizes, that is, 1 to 128, and matches the enumeration 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 transactions, the field must be set to the width of the data channel the transaction is operating across.

Name	`size`
Type	`ARM::AXI::Size`
Getter	`Size get_size() const`
Setter	N/A, fixed at payload construction
RTL signals	ARSIZE, AWSIZE
Default value	N/A, set by the constructor

3.3.4. Burst field

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 the enumeration used by the ARBURST and AWBURST signals.

The `burst` field is optional in the new_payload constructor and defaults to INCR.

The `burst` field is only relevant in read and write operations.

In RTL, the burst type of a snoop transaction is implicitly defined, when constructing a snoop payload, the burst field must be set to INCR, which is the default value.

Name	`burst`
Type	`ARM::AXI::Burst`
Getter	`Burst get_burst() const`
Setter	N/A, fixed at payload construction
RTL signals	ARBURST, AWBURST
Default value	N/A, set by the constructor, defaults to INCR if not specified

3.3.5. Address field

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 the field used by the ARADDR, AWADDR and ACADDR signals.

Name	`address`
Type	`uint64_t`
Getter	`uint64_t get_address() const`
Setter	`void set_address(uint64_t new_address)`
RTL signals	ARADDR, AWADDR, ACADDR
Default value	N/A, set by the constructor

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, therefore, 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.

3.3.6. ID field

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 write operations. In snoop transactions, this field must be ignored.

Name	`id`
Type	`uint32_t`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARID, RID, AWID, BID
Default value	0

3.3.7. Lock field

The `lock` field indicates the exclusiveness of the transaction. It is set to `LOCK_NORMAL` at payload construction and can be altered after construction.

The enumeration matches the enumeration used by the ARLOCK and AWLOCK signals.

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

Name	`lock`
Type	`ARM::AXI::Lock`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARLOCK, AWLOCK
Default value	`LOCK_NORMAL` (0)

3.3.8. Cache field

The `cache` field indicates the Cacheability of the transaction. It is set to `CACHE_AR_DEVICE_NB`/`CACHE_AW_DEVICE_NB` at payload construction and can be altered after construction.

The enumeration matches the enumeration used by the ARCACHE and AWCACHE signals.

The element is accessible as an integer value, an enumeration for read and write transactions. It is also accessible as a set of bit fields using the `CacheBitEnum` type allowing access to the individual B, M, A and AO bits. The field is only relevant in read and write operations. In snoop transactions, it must be ignored.

Name	`cache`
Type	`ARM::AXI::Cache`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARCACHE, AWCACHE
Default value	`CACHE_AR_DEVICE_NB` / `CACHE_AW_DEVICE_NB` (0)

3.3.9. Prot field

The `prot` field indicates the protection level of the transaction. It is set to `PROT_D_S_UP (data, secure, unprivileged)` at payload construction and can be altered after construction.

The enumeration matches the enumeration used by the ARPROT, AWPROT and ACPROT signals.

The element is accessible as an integer value, an enumeration, and as a set of bit fields using the `ProtBitEnum` type allowing access to the individual P, NS, and I bits.

Name	`prot`
Type	`ARM::AXI::Prot`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARPROT, AWPROT, ACPROT
Default value	`PROT_D_S_UP` (0)

3.3.10. QoS field

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 write operations. In snoop transactions, this field must be ignored.

Name	`qos`
Type	`uint8_t`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARQOS, AWQOS
Default value	0

3.3.11. Region field

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 write operations. In snoop transactions, this field must be ignored.

Name	`region`
Type	`uint8_t`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARREGION, AWREGION
Default value	0

3.3.12. User field

The `user` field indicates the user-defined signaling 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 write operations. In snoop transactions, this field should be ignored.

Name	`user`
Type	`uint64_t`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARUSER, RUSER, AWUSER, WUSER, BUSER
Default value	0

3.3.13. Snoop field

The `snoop` field indicates the transaction coherency type. It is set to `NO_SNOOP` at payload construction and can be altered after construction.

The enumeration matches the enumeration used by the ARPROT, ARSNOOP, AWSNOOP, and ACSNOOP signals.

Name	`snoop`
Type	`ARM::AXI::Snoop`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARSNOOP, AWSNOOP, ACSNOOP
Default value	`SNOOP_AW_WRITE_NO_SNOOP` / `SNOOP_AR_READ_NO_SNOOP` / `SNOOP_AC_READ_ONCE` (0)

3.3.14. Domain field

The `domain` field indicates the Shareability domain of the transaction. It is set to `DOMAIN_NSH` (not shared) at payload construction and can be altered after construction.

The enumeration matches the enumeration used by the ARDOMAIN and AWDOMAIN signals.

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

Name	`domain`
Type	`ARM::AXI::Domain`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARDOMAIN, AWDOMAIN
Default value	`DOMAIN_NSH` (0)

3.3.15. Bar field

The `bar` field indicates whether the operation is a barrier transaction. It is set to `BAR_NORM` (normal access) at payload construction and can be altered after construction.

The enumeration matches the enumeration used by the ARBAR and AWBAR signals.

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

Name	`bar`
Type	`ARM::AXI::Bar`
Getter	N/A, directly accessible
Setter	N/A, directly accessible

RTL signals	ARBAR, AWBAR
Default value	`BAR_NORM` (0)

3.3.16. Unique field

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.

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

Name	`unique`
Type	`bool`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	AWUNIQUE
Default value	`false`

3.3.17. Atop field

The `atop` field is used by atomic operations. It is set to `ATOP_NON_ATOMIC` at payload construction and can be altered after construction. The field is only relevant in writes operations.

In read and snoop transactions, this field must be ignored.

Name	`atop`
Type	`ARM::AXI::Atop`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	AWATOP
Default value	`false`

3.3.18. VMID Extension field

The VMID extension field indicates the high order bits of the VMID used in DVM transactions. It is set to 0 at payload construction and can be altered after construction.

The field is only relevant in DVM carrying read and snoop transactions. In write operations, it should be ignored.

Name	`vmid_ext`
Type	`uint8_t`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARVMIDEXT, ACVMIDEXT

Default value	0

3.3.19. Stash NID field

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 construction.

The field is only relevant in write operations. In read and snoop transactions, it should be ignored.

Name	`stash_nid`
Type	`uint16_t`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	AWSTASHNID
Default value	0

3.3.20. Stash NID Valid field

The Stash NID Valid field indicates the Stash NID field is valid and should be used. It is set to false at payload construction and can be altered after construction.

The field is only relevant in write operations. In read and snoop transactions, it should be ignored.

Name	`stash_nid_valid`
Type	`bool`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	AWSTASHNIDEN
Default value	false

3.3.21. Stash LPID field

The Stash LPID field indicates the logical processor sub-unit 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 construction.

The field is only relevant in write operations. In read and snoop transactions, it should be ignored.

Name	`stash_lpid`
Type	`uint8_t`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	AWSTASHLPID
Default value	0

3.3.22. Stash LPID Valid field

The Stash LPID Valid field indicates the Stash LPID field is valid and should be used. It is set to false at payload construction and can be altered after construction.

The field is only relevant in write operations. In read and snoop transactions, it should be ignored.

Name	`stash_lpid_valid`
Type	`bool`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	AWSTASHLPIDEN
Default value	`false`

3.3.23. Resp field

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.

Name	`resp`
Type	`ARM::AXI::Resp`
Getter	`Resp get_resp() const`
Setter	`void set_resp(Resp resp)`
RTL signals	RRESP, BRESP, CRRESP
Default value	`RESP_OKAY` (0)

In read operations, each beat can have a different resp value. If resp values for all beats are equal, resp for the whole payload will be set to that value. If beat resps are not all equal, the resp field will be set to `RESP_INCONSISTENT` and the resp values for each beat will be accessible with the `read_out_resps()` function described in 3.5.1 AXI transaction-level read functions. The field should not be set to `RESP_INCONSISTENT` directly.

3.3.24. Unique ID field

The unique ID indicator is an optional flag that indicates when a request on the read and write address channels uses an AXI identifier that is unique for in-flight transactions. It is set to false at payload construction and can be altered after construction.

The field is only relevant in read and write operations. In snoop transactions, it should be ignored.

Name	`idunq`
Type	`bool`
Getter	N/A, directly accessible

Setter	N/A, directly accessible
RTL signals	ARIDUNQ, RIDUNQ, AWIDUNQ, BIDUNQ
Default value	false

3.3.25. Chunk Enable field

The read data chunking option enables a Target interface to send read data for a transaction in any order using a 128-bit granule. It is set to false at payload construction and can be altered after construction.

The field is only relevant in read operations. In write and snoop transactions, it should be ignored.

Name	`chunk_en`
Type	`bool`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARCHUNKEN
Default value	false

3.3.26. Untranslated Transaction Secure Stream Identifier field

In untranslated transactions, this field represents the Secure Stream Identifier. When deasserted it indicates a Non-secure stream. When asserted it indicates a Secure stream. It is set to false at payload construction and can be altered after construction.

Name	`mmu_sec_sid`
Type	`bool`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARMMUSECSID, AWMMUSECSID
Default value	false

3.3.27. Untranslated Transaction Secure Identifier field

In untranslated transactions, this field represents the Stream Identifier. Secure and Non-secure streams use different namespaces, qualified with AxMMUSECSID, so they can use the same stream identifier values. It is set to 0 at payload construction and can be altered after construction.

Name	`mmu_sid`
Type	uint32_t
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARMMUSID, AWMMUSID
Default value	0

3.3.28. Untranslated Transaction Substream Identifier Valid field

In untranslated transactions, this field represents the Substream Identifier Valid. Indicates that the transaction has a substream identifier. When deasserted, this signal indicates that the transaction does not have a substream identifier. When asserted, this signal indicates that the transaction has a substream identifier. It is set to false at payload construction and can be altered after construction.

Name	`mmu_ssid_v`
Type	bool
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARMMUSSIDV, AWMMUSSIDV
Default value	false

3.3.29. Untranslated Transaction Substream Identifier field

In untranslated transactions, this field represents the Substream Identifier. This signal is only valid if AxMMUSSIDV is asserted. For a single stream, the stream with substream 0 is a different stream from the stream with no valid substream. It is set to 0 at payload construction and can be altered after construction.

Name	`mmu_ssid`
Type	uint32_t
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARMMUSSID, AWMMUSSID
Default value	0

3.3.30. Untranslated Transaction Address Translated field

In untranslated transactions, this field represents Address Translated. It indicates that the transaction has already undergone PCIe ATS translation. This translation might be a full or partial translation in cases where two stages of translation are supported. When deasserted, this signal indicates that the transaction has not been translated. When asserted, this signal indicates that the transaction has been translated. It is set to false at payload construction and can be altered after construction.

Name	`mmu_atst`
Type	bool
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARMMUATST, AWMMUATST
Default value	`false`

3.3.31. Tag Match field

The Tag Match field indicates the result of a tag comparison on a write transaction. It is set to false at payload construction and can be altered after construction. It represents BTAGMATCH[0] in RTL. BTAGMATCH[1] is encoded into the phase. The field is only relevant in write operations. In read and snoop transactions, it should be ignored.

Name	`tag_match`
Type	`bool`
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	BTAGMATCH[0]
Default value	`false`

3.3.32. TagOp field

The TagOp field indicates the operation to be performed on the tags. It is set to TAG_OP_INVALID at phase construction.

Name	`tag_op`
Type	TagOp
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARTAGOP, AWTAGOP
Default value	TAG_OP_INVALID

3.3.33. CMO field

The CMO field indicates the operation indicates the type of operation being requested. It is set to CMO_CLEAN_INVALID at phase construction.

Name	cmo
Type	CMO
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	AWCMO
Default value	CMO_CLEAN_INVALID

3.3.34. MPAM field

The MPAM field contains the Memory Partitioning and Monitoring (MPAM) information. It is initialized to the default constructor at payload construction.

Name	`mpam`
Type	Mpam

Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARMPAM, AWMPAM
Default value	Mpam()

3.3.35. NSAID field

The NSAID field contains the Non-secure access identifiers. It is initialized to 0 at payload construction.

Name	`nsaid`
Type	uint8_t
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARNSAID, AWNSAID, CRNSAID
Default value	0

3.3.36. Loopback field

The Loopback field permits an agent that is issuing transactions to store information that is related to the transaction in an indexed table. It is initialized to 0 at payload construction.

Name	`loop`
Type	uint64_t
Getter	N/A, directly accessible
Setter	N/A, directly accessible
RTL signals	ARLOOP, AWLOOP, RLOOP, BLOOP
Default value	0

3.4. AXI helper functions

Several additional helper functions are provided to help working with AXI payloads.

`get_beat_count()`

As 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_count`
Prototype	`unsigned get_beat_count() const`

`get_beats_complete()`

The `get_beats_complete()` function provides the number of beats that have been filled within the payload. Models should not expect to be able to access beats which have not yet been communicated. This function is provided for debug purposes.

Function	`get_beats_complete`
Prototype	`unsigned get_beats_complete() const`

`get_data_length()`

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 returned by `get_data_length()`.

Function	`get_data_length`
Prototype	`std::size_t get_data_length() const`

`get_base_address()`

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.

Function	`get_base_address`
Prototype	`uint64_t get_base_address() const`

`get_atomic_response_length()`

In Atomic transactions, `get_atomic_response_length()` gives the length of the response.

Function	`get_atomic_response_length`
Prototype	`std::size_t get_atomic_response_length() const`

`get_atomic_response_beat_count()`

In Atomic transactions, `get_atomic_response_beat_count()` gives the number of response data beats.

Function	`get_atomic_response_beat_count`
Prototype	`unsigned get_atomic_response_beat_count() const`

`get_atomic_response_beat_length ()`

In Atomic transactions, `get_atomic_response_beat_length()` gives the length of a single beat.

Function	`get_atomic_response_beat_length`
Prototype	`std::size_t get_atomic_response_beat_length() const`

`get_unaligned_skipped_chunks ()`

In chunking transactions, `get_unaligned_skipped_chunks()` gives the number of chunks which are skipped. Skipped chunks are within the payload, occupy the lowest chunk indexes, but are not valid to be accessed.

Function	`get_unaligned_skipped_chunks`

Prototype	`unsigned get_unaligned_skipped_chunks() const`

3.5. AXI transaction-level data and response functions

Unlike the Generic TLM 2.0 payload, in `ARM::AXI::Payload` the data 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 into the payload.

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

3.5.1. AXI transaction-level read functions

The library provides a set of read functions for whole read transactions.

`read_in()`

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 that is `get_beat_count() Resps` long.

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

`read_out()`

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

Function	`read_out`
Prototype	`void read_out(uint8_t* data) const`

`read_out_resps()`

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

Function	`read_out_resps`
Prototype	`void read_out_resps(Resp* resp) const`

3.5.2. AXI transaction-level write functions

The library provides a set of write functions for whole write transactions.

`write_in()`

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 what bytes to write. The strobes are organized as one bit for each data byte with the lowest bit

of byte `strobe[0]` corresponding to the lowest byte of data. If `strobe` is `NULL` (that is, the default value if not specified), all bytes are written.

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

`write_out()`

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

Function	`write_out`
Prototype	`void write_out(uint8_t* data) const`

`write_out_strobes()`

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

Function	`write_out_strobes`
Prototype	`void write_out_strobes(uint8_t* strobe) const`

3.5.3. AXI transaction-level snoop functions

The library provides a set of snoop functions for whole snoop transactions.

`snoop_in()`

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

Function	`snoop_in`
Prototype	`void snoop_in(const uint8_t* data)`

`snoop_out()`

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

Function	`snoop_out`
Prototype	`void snoop_out(uint8_t* data) const`

3.6. AXI beat-level data and response functions

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 once that beat has communicated across the socket. The beat index supplied to the copy out access functions is the index of the beat in communication sequence, not address sequence.

3.6.1. AXI beat-level read functions

The library provides a set of read functions for beat-level read transactions.

`read_in_beat()`

Copy in the data for one beat of a read transaction from an array `get_beat_data_length()` bytes long. The beat response will be set to `resp`.

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

`read_out_beat()`

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 which have already been written into a payload can be read out.

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

`read_out_beat_resp()`

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

Function	`read_out_beat_resp`
Prototype	`Resp read_out_beat_resp(unsigned beat_index) const`

3.6.2. AXI beat-level write functions

The library provides a set of write functions for beat-level write transactions.

`write_in_beat()` for an array

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 data of the whole transaction.

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

`write_in_beat()` for a beat shorter than 64 bytes

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

`write_out_beat()`

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 which have already been written into a payload can be read out.

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

`write_out_beat_strobe()` for an array

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 which 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	`void write_out_beat_strobe(unsigned beat_index, uint8_t* strobe) const`

`write_out_beat_strobe()` for a beat shorter than 64 bytes

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 will be organized with the lowest bit corresponding to the lowest address byte in the beat.

Function	`write_out_beat_strobe`
Prototype	`uint64_t write_out_beat_strobe(unsigned beat_index) const`

3.6.3. AXI beat-level snoop functions

The library provides a set of snoop functions for beat-level snoop transactions.

`snoop_in_beat()`

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

Function	`snoop_in_beat`
Prototype	`void snoop_in_beat(const uint8_t* data)`

`snoop_out_beat()`

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 which have already been written into a payload can have their strobe copied out.

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

3.7. AXI chunk-level data and response functions

The library provides a set of functions for chunk-level transactions.

`read_in_chunk()`

Copy in the data for a single chunk to an array of 16 bytes.

Function:	`read_in_chunk`
Prototype:	`void read_in_chunk(unsigned chunk_number, const uint8_t* data, Resp resp = RESP_OKAY)`

`read_out_chunk()`

Copy out the data for a single chunk to an array of 16 bytes.

Function:	`read_out_chunk`
Prototype:	`void read_out_chunk(unsigned chunk_number, uint8_t* data) const`

`read_out_chunk_resp()`

Get the response for a single chunk.

Function:	`read_out_chunk_resp`
Prototype:	`Resp read_out_chunk_resp(unsigned chunk_number) const`

3.8. AXI atomic response data functions

Data associated with the request for an atomic transaction is accessible using the standard write functions. The atomic response is accessible in the same payload using a set of functions for atomic transactions that the library provides.

`read_in_atomic_response()`

Copy in the data for a whole atomic transaction response to an array `get_atomic_response_length()` bytes.

Function:	`read_in_atomic_response`
Prototype:	`void read_in_atomic_response(const uint8_t* data)`

`read_out_atomic_response()`

Copy in the data for a whole atomic transaction response to an array `get_atomic_response_length()` bytes.

Function:	`read_out_atomic_response`
Prototype:	`void read_out_atomic_response(const uint8_t* data)`

`read_in_atomic_response_beat()`

Copy in the data for one beat of an atomic transaction response to an array `get_atomic_response_beat_length()` bytes.

Function:	`read_in_atomic_response_beat`
Prototype:	`void read_in_atomic_response_beat (const uint8_t* data)`

`read_out_atomic_response_beat()`

Copy out the data for one beat of an atomic transaction response to an array `get_atomic_response_beat_length()` bytes.

Function:	`read_out_atomic_response_beat`
Prototype:	`void read_out_atomic_response_beat (const uint8_t* data)`

3.9. AXI Memory Tagging Extension functions

The library provides a set of functions for using the Memory Tagging Extension (MTE).

`get_mte_tag_count()`

Get the number of MTE tags needed to be sent with the transaction.

Function:	`get_mte_tag_count`
Prototype:	`unsigned get_mte_tag_count() const`

`set_mte_tag()`

Set the MTE tag at a given index.

Function:	`set_mte_tag`
Prototype:	`void set_mte_tag(unsigned chunk_index, MteTag tag)`

`get_mte_tag()`

Get the MTE tag at a given index.

Function:	`get_mte_tag`
Prototype:	`MteTag get_mte_tag(unsigned chunk_index) const`

3.10. AXI signal-level support functions

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

3.10.1. AXI signal-level support read functions

The library provides a set of read functions to support signal-level read transactions.

`read_in_beat_raw()`

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

Function: `read_in_beat_raw`

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

`read_out_beat_raw()`

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

Function: `read_out_beat_raw`

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

`read_in_chunk_beat_raw()`

Copy in the data for one beat of a chunking read transaction supplied in raw signal level format. `width` is the width of the data channel (in bytes) in the signal level implementation of the AXI port. `data` is an array (`1 << width`) bytes long. `chunk_number` contains the index of the chunk number, as supplied by RCHUNKNUM in signals or Phase. `chunk_strobe` contains the strobe , as supplied by RCHUNKNUM in signals or Phase.

Function: `read_in_chunk_beat_raw`

Prototype:
```
void read_in_chunk_beat_raw(Size width, const uint8_t*
data, unsigned chunk_number, unsigned chunk_strobe, Resp
resp = RESP_OKAY)
```

`read_out_chunk_beat_raw()`

Copy out the data for one beat of a read transaction supplied in raw signal level format. `width` is the width of the data channel (in bytes) in the signal level implementation of the AXI 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.

Function: `read_out_chunk_beat_raw`

Prototype:
```
void read_out_chunk_beat_raw(Size width, unsigned
chunk_number, unsigned chunk_strobe, uint8_t* data)
const
```

3.10.2. AXI signal-level support write functions

The library provides a set of write functions to support signal-level write transactions.

`write_in_beat_raw()` for a data channel

Copy in the data for one beat of a write transaction supplied in raw signal level format. `width` is the width of the data channel (in bytes) in the signal level implementation of the AXI 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 data of the whole transaction.

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

`write_in_beat_raw()` for a data channel under 64 bytes wide

Copy in the data for one beat of a write transaction 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 AXI 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_in_beat_raw`
Prototype:	`void write_in_beat_raw(Size width, const uint8_t* data, uint64_t strobe)`

`write_out_beat_raw()`

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

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

`write_out_beat_raw_strobe()`

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 AXI 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 data of the whole transaction.

Function:	`write_out_beat_raw_strobe`
Prototype:	`void write_out_beat_raw_strobe(Size width, unsigned beat_index,`

`write_out_beat_raw_strobe()` for a data channel under 64 bytes wide

Get the strobe for one beat of a write transaction 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 AXI 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.

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

3.10.3. AXI signal-level support snoop

There are no specific signal-level helper functions for handling snoop data. You can use the signal-level read functions to handle the CD AXI channel.

3.11. AXI payload propagation

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

In a very 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 that only propagates the transaction unchanged to forward the payload that was received.

Other models need to 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. No other fields should be changed as the payload can still be in use by other models.

3.11.1. AXI descend function

The `descend()` function creates a new payload and it copies all fields from a previous payload to create a new 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 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.

We recommend you use `descend()` when creating derived payloads rather than directly creating new payloads. See 5.1.1 UID field and 5.1.2 Parent field for features that can help when handling derived payloads. When responses of descended transactions return, it is the job of the model that descended the payload to copy the response and data from the descended payload to the original before propagating the original back upstream.

3.11.2. AXI clone function

`clone()` creates a new payload based on the original, similar to `descend()`, but data and response fields of the payload are shared between the cloned payload and the original.

Models such as simple interconnects, address mappers and other models that change transaction request fields but do not alter the command or the burst qualifications should use the `clone()` function to create a new payload.

A new payload created using `clone()` will have all fields set to the same values as the original. Construction time fields (command, size, len, burst and bits of the address) cannot be changed. All other fields can be changed before the payload is propagated.

As 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. The model which cloned the payload does not need to copy the transaction response between the cloned and original.

4. CHI protocol

The AMBA CHI TLM 2.0 Library is designed to be sufficiently flexible to enable modeling of any CHI connected element. This includes coherency and caching agents. The CHI payload is designed to be usable by a range of abstractions from purely behavioral untimed models, to interfacing with RTL signal level simulations at a signal and cycle-accurate level.

4.1. CHI phases

CHI communications use several channels.

CHI channels

Each CHI port logically consists of several CHI channels. The complete set of channels is:

- REQ
- SNP
- RSP
- DAT

Individual channels can be replicated to form multiple subchannels.

CHI channel phases

The channels of a port are modeled with a single TLM 2.0 socket. The channel field in `ARM::CHI::Phase` indicates what channel is being used.

CHI uses link credits for flow control. With non-blocking communication, flits sent using a link credit are accepted by the receiver without requiring a response. For example, `nb_transport_fw/bw()` returns with `tlm::TLM_ACCEPTED`. For more information on link credits, see 4.3 CHI link credits.

4.2. CHI payload alterations

As with other TLM APIs, fields within the CHI TLM payload are considered one time mutable. A field is set once by the relevant agent (either the requestor, or the target) and must not change again for the duration of the lifetime of the transaction. As different parts of the transaction (requests, responses, and data) can be communicated at different points in the system simultaneously, changing the payload will result in changing all views of the payload globally, which is generally incorrect.

Due to CHI interconnects not being restricted to acyclic directed graph formation, the path of the response can be different than the path of the request. In traditional schemes (for example, AXI) it was possible to change fields of a payload request, by creating a new payload with the new field values and passing this downstream. On the receipt of a response, the field altering agent can propagate any response fields to the original payload and use this as the upstream response. As the response in the CHI might take a different path than the request, and therefore the opportunity to convert the descended payload back to the original before forwarding it to the requestor is no longer available.

The payload descending field alteration scheme can only be used if the altering agent is the only route to a requestor or target.

Many of the CHI fields change several times during the lifetime of the transaction. For example, the TgtID, used by the interconnect to route the flit to the correct destination, will be different in various flits (request, response, and data). Some transactions might generate two simultaneous responses to different targets. As altering the payload post is not permitted, such fields are implemented in the phase object. The phase is only valid during the TLM calls and a copy made in case the content is required after a return from this call type.

4.3. CHI link credits

Link credits are communicated through TLM calls. These calls can set the appropriate Channel, SubChannel values, and set LCrd to true in the phase object or set the command to LCrdReturn to communicate a return of a credit. The payload used should not be altered by the target because no fields within it are valid. The Requester can use either a valid payload or use the dummy payload.

4.4. CHI payload generation

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

A new CHI payload is constructed by calling `ARM::CHI::Payload::new_payload()`.

Unless otherwise stated, newly generated payloads will have all fields set to 0, except for the AllowRetry field.

The payload is allocated with a reference count of 1. The reference count of the payload can be incremented and decremented using the `ref()` and `unref()` functions respectively. It is the duty of the model that called the constructor to `unref()` the payload once it is no longer holding any pointers to it. All models which keep a pointer to a payload should `ref()` the payload and `unref()` it once the pointer is discarded.

4.5. CHI payload use

All requests use a new payload unless the CHI protocol requires the request to generate responses to a different node. If responses to a different node are required, the request must use the same payload as the request that the responses belong to.

For example, all requests generated by a Request Node use a new payload, but a Home Node must use an existing payload if it chooses to complete a transaction using DMT, DCT, or DWT.

In CHI, messages that are not attributable to a specific transaction can:

- Use a new payload.

- Use an existing payload, if fields in that payload match payload fields in the message to send or can be updated, subject to the payload alteration rules in 4.2 CHI payload alterations.
- Use the dummy payload if the message does not carry any fields in the payload.

Messages that this payload applies to include:

- L-Credits
- PCrdGrant
- *LCrdReturn
- TagMatch
- Persist

4.6. CHI field location summary

For CHI, the fields are split between phase and payload.

The following table shows each CHI field and its location and the corresponding TLM field.

Table 4-1: CHI field locations

CHI field	Location	TLM field
Target Identifier, TgtID	Phase	`tgt_id`
Source Identifier, SrcID	Phase	`src_id`
Home Node Identifier, HomeNID	Phase	`home_nid`
Return Node Identifier, ReturnNID	Phase	`return_nid`
Forward Node Identifier, FwdNID	Phase	`fwd_nid`
Logical Processor Identifier, LPID	Payload	`lpid`
Persistence Group Identifier, PGroupID	Payload	`p_group_id`
Stash Node Identifier, StashNID	Phase	`stash_nid`
Stash Node Identifier Valid, StashNIDValid	Payload	`stash_nid_valid`
Stash Logical Processor Identifier, StashLPID	Phase	`stash_lpid.value`
Stash Logical Processor Identifier Valid, StashLPIDValid	Phase	`stash_lpid.valid`
Stash Group Identifier, StashGroupID	Payload	`stash_group_id`
Transaction Identifier, TxnID	Phase	`txn_id`
Return Transaction Identifier, ReturnTxnID	Phase	`return_txn_id`
Forwarding Transaction Identifier, FwdTxnID	Phase	`fwd_txn_id`
Data Buffer Identifier, DBID	Phase	`dbid`

CHI field	Location	TLM field
Channel opcodes, Opcode	Phase	• req_opcode • snp_opcode • rsp_opcode • dat_opcode • raw_opcode
Deep persistence, Deep	Payload	deep
Address, Addr	Payload	address
Non-secure, NS	Payload	ns
Size of transaction data, Size	Payload	size
Memory Attribute, MemAttr	Payload	mem_attr
Snoop Attribute, SnpAttr	Phase	snp_attr
Do Direct Write Transfer, DoDWT	Phase	do_dwt
Likely Shared, LikelyShared	Payload	likely_shared
Ordering requirements, Order	Phase	order
Exclusive, Excl	Payload	exclusive
Endian	Payload	endian
Allow Retry, AllowRetry	Phase	allow_retry
Expect Completion Acknowledge, ExpCompAck	Phase	exp_comp_ack
SnoopMe	Payload	snoop_me
Return to Source, RetToSrc	Payload	ret_to_src
Data Pull, DataPull	Payload	data_pull
Do not transition to SD state, DoNotGoToSD	Payload	do_not_go_to_sd
Quality of Service priority level, QoS	Phase	qos
Protocol Credit Type, PCrdType	Phase	pcrd_type
Tag Operation, TagOp	Phase	tag_op
Tag	Payload	tag
Tag Update, TU	Payload	tu
Tag Group Identifier, TagGroupID	Payload	tag_group_id
Memory System Performance Resource Partitioning and Monitoring, MPAM	Payload	mpam
Virtual Machine Identifier Extension, VMIDExt	Phase	vmid_ext

4.7. CHI phase fields

The library provides a set of CHI phase fields.

CHI fields are split between the phase and payload, see the table in 4.6 CHI field location summary.

4.7.1. Channel field

The `channel` field indicates the channel the transaction is being communicated on. It is set to CHANNEL_REQ at phase construction. Other permissible values are CHANNEL_SNP, CHANNEL_RSP and CHANNEL_DAT.

Name:	`channel`
Type:	`Channel`
Default value:	`CHANNEL_REQ`

4.7.2. SubChannel field

When channels are replicated on a single interface, the `sub_channel` field indicates the replicated sub-channel index the transaction is being communicated on. It is set to 0 at phase construction.

Name:	`sub_channel`
Type:	`uint8_t`
Default value:	`0`

4.7.3. Opcode field

The `opcode` field specifies the transaction type and is the primary field that determines the transaction structure. It is set to 0 at phase construction. Four different enumerations are used depending on the channel type, only one of which is valid at one time. The `raw_opcode` value allows access to the numerical form of the value.

Name:	`req_opcode, snp_opcode, rsp_opcode, dat_opcode, raw_opcode`
Type:	`ARM::CHI::{Req, Snp, Rsp, Dat}Opcode, uint8_t`
Default value:	`ARM::CHI::{REQ, SNP, RSP, DAT}_OPCODE_REQ_LCRD_RETURN, 0`

4.7.4. LCrd field

The `lcrd` field indicates the call is to pass a link credit. On a credit pass call, only `channel` and `sub_channel` phase fields are valid. All other phase fields and payload fields are undefined. It is set to false at phase construction.

Name:	`lcrd`
Type:	`bool`
Default value:	`false`

4.7.5. SrcID field

The `src_id` field indicates the node ID of the port on the component from which the packet was sent. It is set to 0 at phase construction.

Name:	`src_id`

Type:	`uint16_t`
Default value:	`0`

4.7.6. TgtID field

The `tgt_id` field indicates the node ID of the port on the component to which the packet is targeted. It is set to 0 at phase construction.

Name:	`tgt_id`
Type:	`uint16_t`
Default value:	`0`

4.7.7. StashNID field

The `stash_nid` field indicates the node ID of the Stash target. It is set to 0 at phase construction.

Name:	`stash_nid`
Type:	`uint16_t`
Default value:	`0`

4.7.8. ReturnNID field

The `return_nid` field indicates the node ID that the response with Data is to be sent to. It is set to 0 at phase construction.

Name:	`return_nid`
Type:	`uint16_t`
Default value:	`0`

4.7.9. FwdNID field

The `fwd_nid` field indicates the node ID of the original Requester. It is set to 0 at phase construction.

Name:	`fwd_nid`
Type:	`uint16_t`
Default value:	`0`

4.7.10. HomeNID field

The `home_nid` field indicates the node ID of the target of the CompAck response to be sent from the Requester. It is set to 0 at phase construction.

Name:	`home_nid`
Type:	`uint16_t`
Default value:	`0`

4.7.11. SLCRepHint field

The `slc_rep_hint` field indicates node ID of the Stash target. It is set to 0 at phase construction.

Name:	`slc_rep_hint`
Type:	`uint16_t`
Default value:	0

4.7.12. TxnID field

The `txn_id` field indicates the unique transaction ID for each source node. It is set to 0 at phase construction.

Name:	`txn_id`
Type:	`uint16_t`
Default value:	0

4.7.13. DBID field

The `dbid` field indicates the ID to be used as the TxnID in the response to this message. It is set to 0 at phase construction.

Name:	`dbid`
Type:	`uint16_t`
Default value:	0

4.7.14. ReturnTxnID field

The `return_txn_id` field indicates the unique transaction ID that conveys the value of TxnID in the data response from the Target. It is set to 0 at payload construction. The field is only relevant in DMT transactions.

Name:	`return_txn_id`
Type:	`uint16_t`
Default value:	0

4.7.15. FwdTxnID field

The `fwd_txn_id` field indicates the transaction ID used in the Request by the original Requester. It is set to 0 at payload construction. The field is only relevant in DMT transactions.

Name:	`fwd_txn_id`
Type:	`uint16_t`
Default value:	0

4.7.16. VMIDExt field

The `vmid_ext` field indicates the Virtual Machine ID Extension. It is set to 0 at payload construction.

Name:	`vmid_ext`
Type:	`uint8_t`
Default value:	0

4.7.17. StashLPID field

The `stash_lpid` field indicates the Stash Logical Processor ID value and the Valid bit. Both the value and the valid bit are set to 0 at payload construction.

Name:	`stash_lpid`
Type:	`ARM::CHI::Phase::StashLPID`
Default value:	{0,0}

4.7.18. Order field

The `order` field indicates the order of the transaction. The type holds the same enumeration as the AMBA® 5 CHI Architecture Specification. It is set to `ORDER_NO_ORDER` at phase construction.

Name:	`order`
Type:	`Order`
Default value:	`ORDER_NO_ORDER`

4.7.19. SnpAttr field

The `snp_attr` field indicates the snoop attributes associated with the transaction. It is set to `false` at phase construction.

Name:	`snp_attr`
Type:	`bool`
Default value:	`false`

4.7.20. AllowRetry field

The `allow_retry` field indicates if the target is permitted to give a Retry response. It is set to `true` at phase construction.

Name:	`allow_retry`
Type:	`bool`
Default value:	`true`

4.7.21. DataID field

The `data_id` field indicates the address offset of the data provided in the packet. It is set to 0 at phase construction.

Name:	`data_id`
Type:	`uint8_t`
Default value:	`0`

4.7.22. Resp field

The `resp` field indicates the cache line state associated with a data transfer. It is set to `RESP_I` at phase construction.

Name:	`resp`
Type:	`Resp`
Default value:	`RESP_I`

4.7.23. FwdState field

The `fwd_state` field indicates the cache line state associated with a data transfer to the Requester from the receiver of the snoop. It is set to `RESP_I` at phase construction.

Name:	`fwd_state`
Type:	`Resp`
Default value:	`RESP_I`

4.7.24. RespErr field

The `resp_err` field indicates the error status associated with a data transfer. It is set to `RESP_ERR_OK` at phase construction.

Name:	`resp_err`
Type:	`RespErr`
Default value:	`RESP_ERR_OK`

4.7.25. CBusy field

The `c_busy` field indicates the current level of activity at the Completer. It is set to 0 at phase construction.

Name:	`c_busy`
Type:	`uint8_t`
Default value:	`0`

4.7.26. ExpCompAck field

The `exp_comp_ack` bit indicates that the transaction will include a Completion Acknowledge message. It is set to false at payload construction. In snoop transactions, this field must be ignored.

Name:	`exp_comp_ack`
Type:	`bool`
Default value:	`false`

4.7.27. PCrdType field

The `pcrd_type` field indicates the type of Protocol Credit being used by a request that has the AllowRetry field deasserted. It is set to 0 at phase construction.

Name:	`pcrd_type`
Type:	`uint8_t`
Default value:	`0`

4.7.28. QoS field

The `qos` field specifies 1 of 16 possible priority levels for the transaction with ascending values of QoS indicating higher priority levels. It is set to 0 at phase construction.

Name:	`qos`
Type:	`uint8_t`
Default value:	`0`

4.7.29. TagOp field

The `tag_op` field indicates the operation to be performed on the tags present in the corresponding DAT channel. It is set to 0 at phase construction.

Name:	`tag_op`
Type:	`TagOp`
Default value:	`TAG_OP_INVALID`

4.7.30. DoDWT field

The `do_dwt` field indicates that the Subordinate requests write data directly from the Requester. It is set to false at payload construction.

Name:	`do_dwt`
Type:	`bool`
Default value:	`false`

4.8. CHI payload fields

The payload can represent any CHI transaction.

The payload contains fields that use the same enumeration encodings as the hardware implementation as listed in the AMBA® 5 CHI Architecture Specification.

CHI fields are split between the phase and payload, see the table in 4.6 CHI field location summary.

4.8.1. Payload address field

The `address` field indicates the address value of the transaction. It is set to 0 at payload construction.

Name:	`address`
Type:	`uint64_t`
Default value:	`0`

When transmitting data on channels smaller than a cache line, the critical chunk (that is, the beat containing the byte that is addressed) is decoded from the respective bits of the address field. For example, when addressing a 64-byte cache line on a 16-byte data channel, bits 5:4 are assumed to be the critical chunk ID (CCID). This chunk is, when possible, given transmission priority and is encoded as DataID in the DAT phase.

4.8.2. Size field

The `size` field indicates the Size value of the transaction. The enumeration matches that used by the `size` field used in the REQ channel. It is set to `SIZE_1` at payload construction. In snoop transactions, this field must be ignored.

Name:	`size`
Type:	`ARM::CHI::Size`
Default value:	`SIZE_1 (0)`

4.8.3. MemAttr field

The `mem_attr` field indicates the MemAttr value of the transaction. The enumeration matches that used by the MemAttr field used in the REQ channel. It is set to 0 at payload construction. In snoop transactions, this field must be ignored.

Name:	`mem_attr`
Type:	`ARM::CHI::MemAttr`
Default value:	`MEM_ATTR_NORMAL_NC_NB (0)`

4.8.4. LPID field

The `lpid` field indicates the Logical Processor ID (LPID), used in conjunction with the `src_id` field to uniquely identify the logical processor that generated the request. It is set to 0 at payload construction. In snoop transactions, this field must be ignored.

Name:	`lpid`
Type:	`uint8_t`
Default value:	`0`

4.8.5. PGroupID field

The `p_group_id` field indicates the set of CleanSharedPersistSep transactions to which the request applies. It is set to 0 at payload construction.

Name:	`p_group_id`
Type:	`uint8_t`
Default value:	`0`

4.8.6. StashGroupID field

The `stash_group_id` field indicates the set of StashOnceSep transactions to which the request applies. It is set to 0 at payload construction.

Name:	`stash_group_id`
Type:	`uint8_t`
Default value:	`0`

4.8.7. TagGroupID field

The `tag_group_id` is typically expected to contain Exception Level, TTBR value, and CPU identifier. It is set to 0 at payload construction.

Name:	`tag_group_id`
Type:	`uint8_t`
Default value:	`0`

4.8.8. Excl field

The `exclusive` field indicates that the corresponding transaction is an Exclusive access transaction. It is set to false at payload construction. In snoop transactions, this field must be ignored.

Name:	`exclusive`
Type:	`bool`
Default value:	`false`

4.8.9. SnoopMe field

The `snoop_me` field indicates that Home must determine whether to send a snoop to the Requester. It is set to false at payload construction. The field is only relevant in Atomic transactions.

Name:	`snoop_me`
Type:	`bool`
Default value:	`false`

4.8.10. Endian field

The `endian` bit indicates the endianness of Data in the Data packet for Atomic transactions. It is set to false at payload construction. The field is only relevant in Atomic operations.

Name:	`endian`
Type:	`bool`
Default value:	`false`

4.8.11. StashNIDValid field

The `stash_nid_valid` field indicates the Stash NID field is valid and should be used. It is set to false at payload construction. The field is only relevant in write operations. In read and snoop transactions, it should be ignored.

Name:	`stash_nid_valid`
Type:	`bool`
Default value:	`false`

4.8.12. Deep field

The `deep` bit indicates that the Persist response must not be sent until all earlier writes are written to the final destination. It is set to false at payload construction. The field is only relevant in Persistent CMO operations.

Name:	`deep`
Type:	`bool`
Default value:	`false`

4.8.13. NS field

The `ns` field indicates if the transaction is Non-secure or Secure. It is set to false at payload construction.

Name:	`ns`
Type:	`bool`
Default value:	`false`

4.8.14. LikelyShared field

The `likely_shared` field indicates an allocation hint for downstream caches. It is set to false at payload construction. In snoop transactions, this field must be ignored.

Name:	`likely_shared`
Type:	`bool`
Default value:	false

4.8.15. RetToSrc field

The `ret_to_src` bit instructs the receiver of the snoop to return Data with the Snoop response. It is set to false at payload construction. The field is only relevant in Snoop transactions.

Name:	`ret_to_src`
Type:	`bool`
Default value:	`false`

4.8.16. DoNotGoToSD field

The `do_not_go_to_sd` bit controls Snoopee use of SD state. It is set to false at payload construction. The field is only relevant in Snoop transactions.

Name:	`do_not_go_to_sd`
Type:	`bool`
Default value:	`false`

4.8.17. RSVDC field

The `rsvdc` field is used for user-defined values. It is set to 0 at payload construction.

Name:	`rsvdc`
Type:	`uint32_t`
Default value:	0

4.8.18. DataPull field

The `data_pull` field indicates the inclusion of an implied read request in the data response.

Name:	`data_pull`
Type:	`ARM::CHI::DataPull`
Default value:	`ARM::CHI:: DATA_PULL_NO_READ`

4.8.19. DataSource field

The `data_source` field indicates the source of the data in a read data response.

Name:	`data_source`
Type:	`uint8_t`
Default value:	0

4.8.20. MPAM field

The `mpam` field contains a label, identifying the partition to which it belongs, together with the performance monitoring group within that partition.

Name:	`mpam`
Type:	`ARM::CHI::Mpam`
Default value:	{0,0,false}

4.8.21. TU field

The `tu` (Tag Update) field indicates which of the allocation tags must be updated.

Name:	`tu`
Type:	`uint8_t`
Default value:	0

4.8.22. Tag field

The `tag` field provides sets of 4-bit tags, each associated with an aligned 16 bytes of data.

Name:	`tag`
Type:	`uint8_t[4]`
Default value:	0

4.8.23. Data field

The `data` field indicates the data of the transaction. It is cleared to 0 at payload construction.

Name:	`data`
Type:	`uint8_t[64]`
Default value:	0

4.8.24. BE field

The `byte_enable` field indicates the byte enable mask of the transaction. It is cleared to 0 at payload construction.

Name:	`byte_enable`
Type:	`uint64_t`
Default value:	0

4.9. CHI payload functions

The library provides a set of payload functions for CHI transactions.

4.9.1. Payload creation functions

The library provides a set of payload creation functions for CHI transactions.

`new_payload()`

Create a new payload. The returned payload has its fields set to their default state.

Function:	`new_payload`
Prototype:	`static Payload* new_payload()`

`descend()`

Create a new payload and setting the parent payload to be the object the function is called on.

Function:	`descend`
Prototype:	`Payload* descend()`

`get_dummy()`

Get the dummy payload that can be used for LCRD passing operations. The content of the payload is undefined. The payload can have ref and unref called on it, but we do not encourage it.

Function:	`get_dummy`
Prototype:	`static Payload* get_dummy ()`

4.9.2. Reference counting functions

The library provides a set of reference counting functions for CHI payloads.

`ref()`

Increment the payload reference count.

Function:	`ref`
Prototype:	`void ref()`

`unref()`

Decrement the payload reference count. If a payload reference count reached zero, it will be returned to the pool.

Function:	`unref`
Prototype:	`void unref()`

4.10. CHI payload propagation

In a very 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 need to 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. No other fields should be changed as the payload can still be in use by other models.

4.10.1. CHI descend function

The `descend()` function creates a new payload and copies all fields from a previous payload to create a new 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 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.

It is strongly encouraged to use `descend()` when creating derived payloads rather than directly creating new payloads. See 5.1.1 UID field and 5.1.2 Parent field for features that can help when handling derived payloads. When responses of descended transactions return, it is the job of the model that descended that payload to copy the response and data from the descended payload to the original before propagating the original back upstream.

5. Additional payload features

Several fields are attached to all payloads to help the programmer working with AXI and CHI payloads. These fields have no representation in the signal-level implementation and should have no influence on the behavior of a model.

5.1. Additional payload fields

The library provides additional payload fields.

5.1.1. UID field

The `uid` field is a unique payload ID that can be used to refer to 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 constructed and is not reused even if a payload is destroyed. `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 and can be used as a sentinel value where `uid` fields are used as key values outside `ARM::AXI::Payload`.

Name:	`uid`
Type:	`uint64_t const`
Getter:	N/A, directly accessible
Setter:	N/A, const
RTL signals:	N/A, simulation construct

5.1.2. Parent field

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 in which case the parent field is set to point to the original payload. It cannot be altered post payload construction.

Descended and cloned payloads that refer to 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.

Name:	`parent`
Type:	`Payload* const`
Getter	N/A, directly accessible
Setter	N/A, const
RTL signals	N/A, simulation construct

5.2. Extension system

The payload extension system allows additional fields to be added to payloads. All payloads have space allocated to hold all extensions whether they are set or not.

Values of extensions are copied when a payload is descended (or cloned in AXI). Extension fields in new payloads are created using the default constructor and destroyed using the default destructor when the payload is returned to the pool. Each extension can be of any object type.

Extensions must be declared before any payload object is created. Because extensions are typically declared by models, no payloads can be created 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 give accessors to the same extension.

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

```
ARM::AXI::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. Reading and writing the value of the extension can be done through this reference. The following example code demonstrates changing an extension value:

```
uint32_t& core_id_value = core_id_accessor.get(&payload);
core_id_value &= 0xF;
```

5.2.1. Custom Extension Managers

When an extension manager is not defined, similar to the previous example, the default extension manager is used.

The default extension manager uses the default constructor, copy constructor, and destructor for the actions of creating a new payload, descending a payload, and returning the payload to the pool, respectively. It is possible to perform relevant actions.

An example of a custom extension manager is available in `tests/extensions.cpp`.

Appendix A. Revisions

This appendix describes the technical changes between released issues of this document.

Table A-1: Issue 01

Change	Location
First issue	-

Table A-2: Differences between issue 01 and issue 02

Change	Location
Document renamed from *Arm® AMBA® TLM 2.0 Library Developer Guide* to *AMBA® TLM 2.0 Library Reference Manual*	-
Previous document issue numbering format updated from 1000-00 to 01	-
Editorial updates	Whole document
Added information about CHI header files and socket types	• 2.1 Header file structure • 2.2 Socket types
Moved section to 2 AMBA TLM 2.0 Library overview	2.3 Clocking
Added sections to 3.1 AXI phases	• 3.1.2 QOSACCEPT • 3.1.3 RCHUNKNUM and RCHUNKSTRB
Added sections to 3 AXI protocol	• 3.1 AXI phases • 3.8 AXI atomic response data functions • 3.9 AXI Memory Tagging Extension functions
Added chapter	4 CHI protocol
Moved sections to 5 Additional payload features	• 5.1 Additional payload fields • 5.2 Extension system