TLM2.0 compliant CHI Transactor Specification

Contents

1 Preface

1.1 About this Specification

This specification details the representation of the CHI protocol in a TLM2.0 compliant implementation. The definition of the protocol adheres to ‘AMBA® CHI™’ Protocol Specification 1.21)

It is assumed that the reader is familiar with the TLM-2.0 language reference manual (see section 1.2) version TLM2.0.1 and has some basic experience with TLM modeling. Basic understanding of the CHI protocol is beneficial.

1.2 References

This manual focuses on the extension of TLM2.0 to model the AMBA CHI protocol at loosely and approximately timed accuracy. For more details on the protocol and semantics, see the following manuals and specifications:

1) AMBA® 5 CHI™ Architecture Specification, 8th May 2018 “IHI0050C_amba_5_chi_architecture_spec-3.pdf”
2) IEEE Std. 1666 TLM-2.0 Language Reference Manual (LRM)

1.3 Revisions

Document Version	Author	Date	Description
0.1	Suresh, Mahendra	20/June/2019	On same lines as AXI TLM spec, Data structures, TLM sockets and architecture broad specification flow
0.2	Suresh, Mahendra	28/June/2019	Added TLM-2.0 extensions, interfaces and socket definitions
0.3	Kaushanski, Stanislaw	8/July/2020	Remove 2nd socket pair, update extension fields, added, functional flow, architecture, phases and transitions diagrams. Updated all chapters except Blocking communication chapter.
0.4	Kaushanski, Stanislaw	21/January/2021	Remove chi_driver implementation and integration info, because it is out of scope of this specification. Updated Blocking chapter. Updated Non-Blocking

			phases. Updated structures and extensions.
0.5	Jentzsch, Eyck	8/February/2021	Correct errors in transaction to channel, socket and phase mapping

2 Introduction

This document specifies the way sockets communicate to each other while modeling properties of the CHI protocol. As such all channels of a CHI interface are represented by a single TLM2.0 socket.
The specification describes the way sockets exchange information not how it is to be implemented.

The various channels used in CHI along with Request Node (RN) channel designation and Slave Node (SN) channel designation is as given below:

Channel	**RN Channel**	**SN Channel**
REQ	TXREQ (Outbound Request)	RXREQ(Inbound Request)
WDAT	TXDAT (Outbound Data)	RXDAT(Inbound Data)
RDAT	RXDAT (Inbound Data)	TXDAT (Outbound Data)
CRSP	RXRSP (Inbound Response)	TXRSP (Outbound Response)
SNP	RXSNP (Inbound Snoop Request)	-
SRSP	TXRSP (Outbound Response)	-

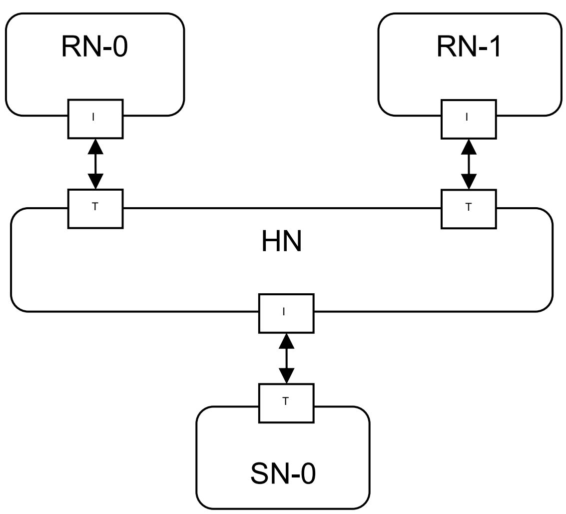

Figure 2. TLM-2.0 Sockets used in CHI System

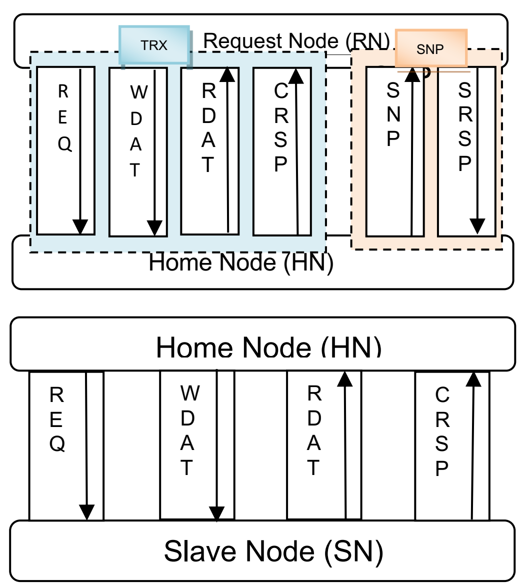

Figure 1. Channels used in CHI System

CHI based system consists of RN (Requester Node), HN (Home Node) and SN (Slave Node) components. In the above figure, a TLM-2.0 based set-up has been taken. The TLM-2.0 based communication between an RN and its connected HN, is done using a pair of TLM-2.0 interface sockets and an initiator-target pair between HN and its connected SN.

- Using the socket-pair between RN and HN,
  (a) the main transaction 'Request' object is sent on the REQ (TXREQ) channel. Typically, a forward non-blocking call 'nb_transport_fw(..)' is used to start the transaction request with BEGIN_REQ phase.
  (b) If transaction requires data to be sent from RN, for ex Write request or Snoop response, then a 'Data' object is sent on the WDAT (TXREQ) channel through the same socket using forward non-blocking path.
  (c) For read operation, multiple 'Data' objects will come on the RDAT channel using the non-blocking backward path.
  (d) The 'Response' object from the Completer (SN or HN) to RN comes on the CRSP channel through the same socket on the non-blocking backward path.

  (a) 'Snoop' request transaction is sent from the HN (as Requester) to the RN-x (as Completer) using backward non-blocking path of the same socket pair.
  (b) It will also send a 'Response' object from the snooped RN-x to the HN, on SRSP channel using the socket forward path.
  (c) The response 'CompAck' from RN to HN, also comes on the SRSP channel and hence using the same forward path.

- All the calls are non-blocking and if transaction starts with BEGIN_REQ on the fwd path, then the call returns with END_REQ phase, as the BEGIN_REQ to END_REQ phase does not need any time. Even if the completer (HN) is busy while taking 'Request', it simply returns with 'RetryAck' so that channel is not blocked and it is tried again by RN when the HN sends a PCrdResponse. Since none of the channels are blocked by any transaction at any stage (request or snoop or response), hence we suggest using non-blocking mechanism only.

- At the link layer level, an L-Credit is expected from Receiver for the Transmitter to send the 'Request' object. The feature is not fully implemented yet. The receiver does not provide the L-Credit value. So in current implementation the number of L-Credits is hardcoded in CHI driver to value of 8.

3 Channel Fields Mapping

The following chapters show the mapping of CHI fields to either the payload, its extansions of phases.
Phases are defined for the non-blocking protocol only. Wherever possible the CHI protocol is mapped to the generic protocol phases to ease interoperability with theTLM2.0 standard. Additional phases are defined in the chapter "Extended Phases"

There are three CHI extension types that are transferred through the various channels to complete a transaction

- 'chi_ctrl_extension' covering control part of access from RN to HN or HN to SN
- 'chi_snp_extension' request transaction object from HN to RN
- 'chi_data_extension' object that carries Write or Read or Snooped data between RN and HN or HN and SN

3.1 Chi_ctrl_extension: extension fields

The 'chi_ctrl_extension' consists of three data structures (common, request and response) and can be used to start a transaction on the 'REQ' channel. This comes from the RN (Requester Node) to the HN (Home Node). This request is made from the "chi_intiator_socket" of the RN (or HN), and handled by the corresponding connected "chi_target_socket" on the HN (or SN).

Field	**where**	**data structure**	**name**	**Type**
Addr	payload	tlm_generic_payload	addr	uint64_t
QoS	extension	chi::common	qos	uint32_t
SrcID	extension	chi::common	src_id	uint16_t
TxnID	extension	chi::common	txn_id	uint16_t
AllowRetry	extension	chi::request	allow_retry	bool
Endian	extension	chi::request	endian	bool
Excl	extension	chi::request	excl	bool
ExpCompAck	extension	chi::request	exp_comp_ack	bool
LikelyShared	extension	chi::request	likely_shared	bool
LPID	extension	chi::request	lp_id	unit8_t
MaxFlit	extension	chi::request	max_flit	unit8_t
MemAttr	extension	chi::request	mem_attr	unit4_t
NS	extension	chi::request	ns	bool
Opcode	extension	chi::request	opcode	req_opcode_e
Order	extension	chi::request	order	uint8_t
PCrdType	extension	chi::request	pcrd_type	uint8_t
ReturnNID	extension	chi::request	return_n_id	uint16_t
ReturnTxnID	extension	chi::request	return_txn_id	uint8_t
RSVDC	extension	chi::request	rsvdc	uint32_t
Size	extension	chi::request	size	uint8_t
SnoopMe	extension	chi::request	snoop_me	bool
snpAttr	extension	chi::request	snp_attr	bool
StashLPID	extension	chi::request	stash_lp_id	uint8_t
StashLPIDValid	extension	chi::request	stash_lp_id_valid	bool

StashNID	extension	chi::request	stash_n_id	uint16_t
StashNIDValid	extension	chi::request	stash_n_id_valid	bool
TgtID	extension	chi::request	tgt_id	uint8_t
TraceTag	extension	chi::request	trace_tag	bool
DBID	extension	chi::response	db_id	uint8_t
PCrdType	extension	chi::response	pcrd_type	uint8_t
RespErr	extension	chi::response	resp_err	uint8_t
FwdState	extension	chi::response	fwd_state	uint8_t
DataPull	extension	chi::response	data_pull	uint8_t
TraceTag	extension	chi::response	trace_trag	bool
TgtID	extension	chi::response	tgt_id	uint16_t
Opcode	extension	chi::response	opcode	rsp_opcode_e
Resp	extension	chi::response	resp	rsp_resptype_e

3.2 Chi_snp_extension: Snooping request fields

The Snoop extension consists of three data structures (common, snp_request and response) and is used for snooping transactions on the SNP channel. Snoop transactions come from the HN (Home Node) to the RN (Requester Node). This request is made from the HN, and handled by the RN.

Field	**where**	**data structure**	**name**	**Type**
Addr	payload	tlm_generic_payload	addr	uint64_t
QoS	extension	chi::common	qos	uint32_t
SrcID	extension	chi::common	src_id	uint16_t
TxnID	extension	chi::common	txn_id	uint16_t
DoNotDataPull	extension	chi::snp_request	do_not_data_pull	bool
DoNotGoToSD	extension	chi::snp_request	do_not_goto_sd	bool
FwdNID	extension	chi::snp_request	fwd_n_id	uint16_t
FwdTxnID	extension	chi::snp_request	fwd_txn_id	uint8_t
NS	extension	chi::snp_request	ns	bool
Opcode	extension	chi::snp_request	opcode	snp_opcode_e
RetToSrc	extension	chi::snp_request	ret_to_src	Bool
StashLPID	extension	chi::snp_request	stash_lp_id	uint8_t
StashLPIDValid	extension	chi::snp_request	stash_lp_id_valid	bool
TraceTag	extension	chi::snp_request	trace_tag	bool
VMIDExt	extension	chi::snp_request	vm_id_ext	uint8_t
DBID	extension	chi::response	db_id	uint8_t
PCrdType	extension	chi::response	pcrd_type	uint8_t
RespErr	extension	chi::response	resp_err	uint8_t
FwdState	extension	chi::response	fwd_state	uint8_t
DataPull	extension	chi::response	data_pull	uint8_t
TraceTag	extension	chi::response	trace_trag	bool
TgtID	extension	chi::response	tgt_id	uint16_t
Opcode	extension	chi::response	opcode	rsp_opcode_e

Resp	extension	chi::response	resp	rsp_resptype_e

3.3 Chi_data_extension: WDATA or RDATA fields

The ‘Data’ extension consists of two data structures (common and data) can be used in the following ways
- sent with the ‘Request’ for Write operation on WDAT (TXDATA) channel
- it can come with the ‘Reponse’ for Read operation on RDAT (RXDATA channel)
- response of snoop operation again on WDAT (TXDATA channel) from snooped master

Field	**where**	**data structure**	**name**	**Type**
BE	payload	tlm_generic_payload	be	unsigned char*
Data	payload	tlm_generic_payload chi::data	data	unsigned char*
QoS	extension	chi::common	qos	uint32_t
SrcID	extension	chi::common	src_id	uint16_t
TxnID	extension	chi::common	txn_id	uint16_t
CCID	extension	chi::data	cc_id	uint8_t
DataCheck	extension	chi::data	data_check	uint64_t
DataID	extension	chi::data	data_id	uint8_t
DataPull	extension	chi::data	data_pull	uint8_t
Datasource	extension	chi::data	data_source	uint8_t
DBID	extension	chi::data	db_id	uint8_t
FwdState	extension	chi::data	fwd_state	uint8_t
HomeNID	extension	chi::data	home_n_id	uint16_t
Opcode	extension	chi::data	opcode	dat_opcode_e
Poison	extension	chi::data	poison	uint8_t
Resp	extension	chi::data	resp	dat_resptype_e
RespErr	extension	chi::data	resp_err	uint8_t
RSVDC	extension	chi::data	rsvdc	uint32_t

4 TLM-2.0 Transaction flow diagram

The following flow-diagram illustrates the usage of TLM-2.0 protocol for two scenarios of transaction, one Snoop-less ‘ReadNoSnp’ and another with Snoop WriteOnce. Each transaction shows the OpCode, type of Transaction Object used and Channel on which it is sent.

4.1 Transaction Flow without Snoop

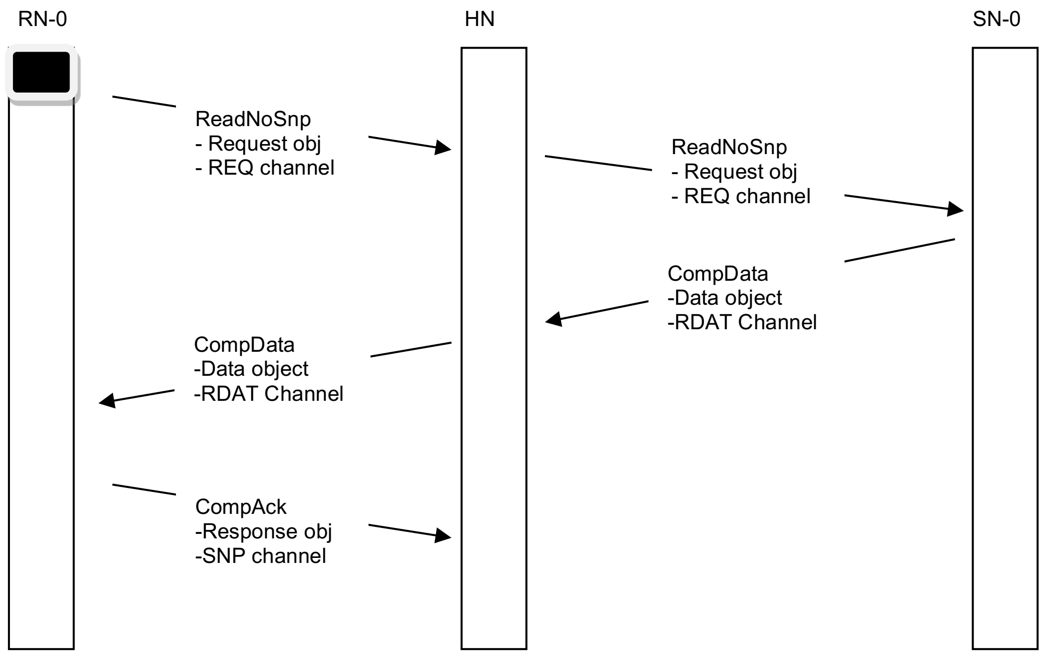

Figure 3 Snoop-less transaction

4.2 Transaction Flow for Snoop-based transaction

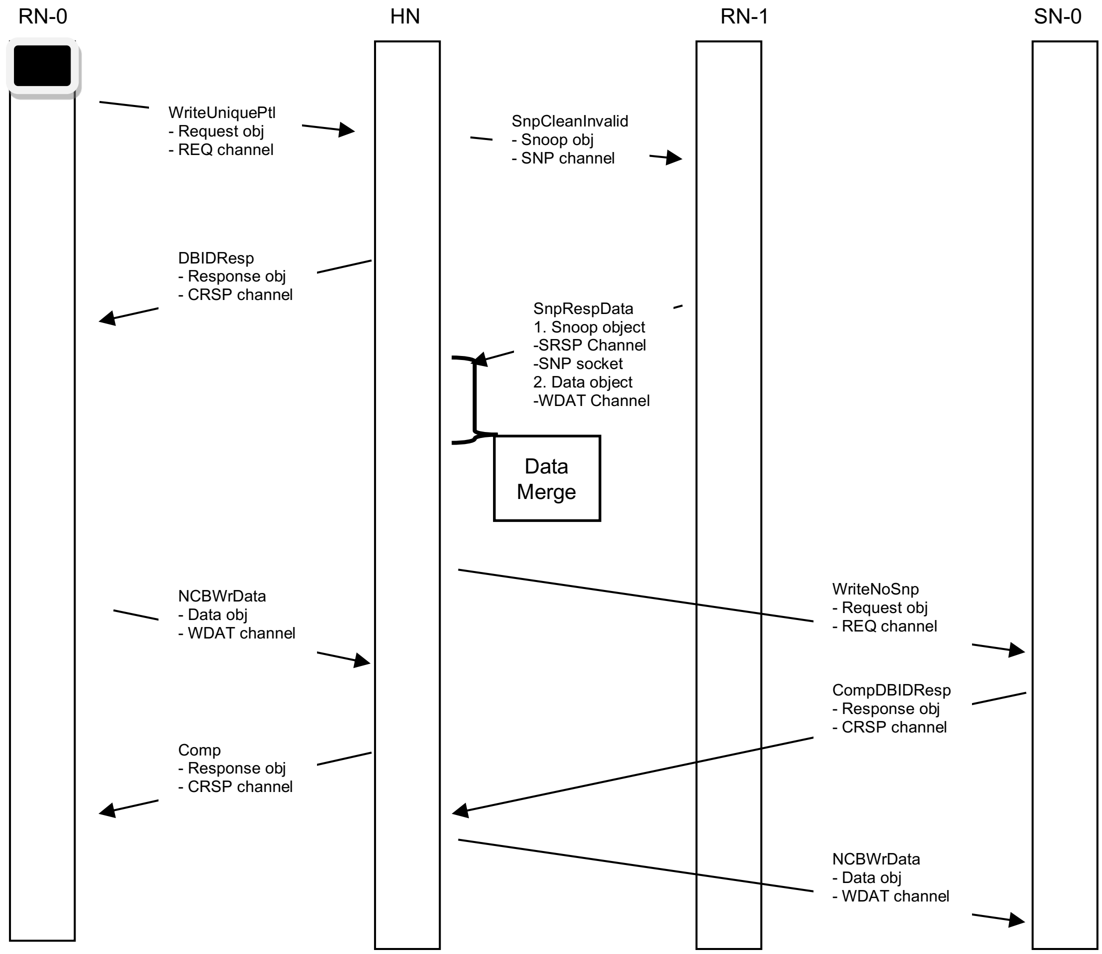

Figure 4 Snoop-based transaction

5 DMI and Debug Transport Communication

The direct memory interface (DMI) and debug transport interface are specialized interfaces distinct from the transport interface, providing direct access and debug access to a resource owned by a target. DMI is intended to accelerate regular memory transactions in a loosely-timed simulation, whereas the debug transport interface is for debug access free of the delays or side-effects associated with regular transactions. For more details on debug transport and DMI please refer to the ‘OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL’.

6 Blocking Communication

Blocking communication is mostly used in loosely-timed (LT) models or programmer view use cases. Here the communication is abstracted and described by 2 timing points: the start and the end of the transaction. CHI sockets use the b_transport interface as described in [5] the TLM-2.0 LRM. For more details on LT modeling, please refer to the ‘OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL’.

6.1 b_transport and b_snoop Call Sequence

The call sequences for blocking transactions are the same than for the generic protocol one. The backward socket interface is been extended to allow for blocking snoop accesses. The semantics of the b_snoop access is the same as the b_transport call but in backward direction.

7 Non-blocking Communication

In the non-blocking communication protocol, each transaction has multiple timing points. This way, the timely description is of higher accuracy and suitable e.g. for architectural exploration.
Each socket interaction is characterized by the generic payload, the phase time points and the direction of communication (forward or backward interface). Therefore, the CHI channels can be identified and it is possible to route them thru the same socket.

7.1 Extended Phases

The non-blocking transactions of the CHI TLM2.0 implementation use up to 7 additional phases:

- BEGIN_PARTIAL_DATA
  Denoting the start of transaction of multiple data packets
- END_PARTIAL_DATA
  Denoting the end of transaction of multiple data packets
- BEGIN_DATA
  Denoting the start of the last data packet in the transaction
- END_DATA
  Denoting the end of the last data packet in the transaction
- ACK
  Denoting the acknowledgement transfer

Following diagram shows the protocol and phase transitions in details:

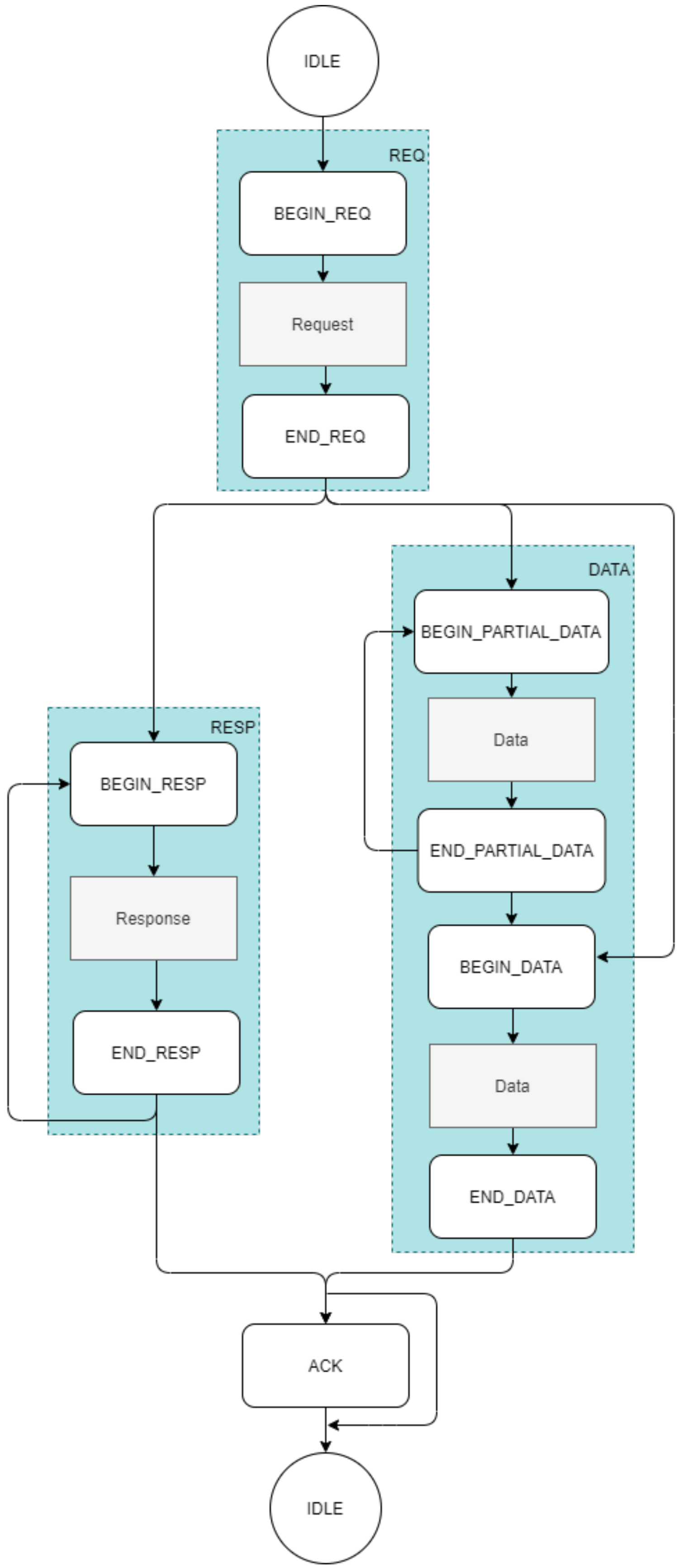

Figure 5: CHI TLM2 protocol phases and transitions

7.2 Transaction to channel, socket and phase mapping

The following table shows the TLM transactions on backward and forward calls, extensions and phases used for doing a transaction call from Requestor to Completer.

S.No.	Transaction	From - To -	Channel	Opcodes	Extension	path	Phase on calling	Return Phase	Status on return
1	Non-snoopable read/write request	RN to HN	REQ	ReadNoSnp	chi_ctrl_extension	Forward	BEGIN_REQ	END_REQ	TLM_UPDATED
								BEGIN_REQ	TLM_ACCEPTED
2	Snoopable read request	RN to HN	REQ	ReadOnce, ReadShared etc	chi_ctrl_extension	Backward	BEGIN_REQ	END_REQ	TLM_UPDATED
								BEGIN_REQ	TLM_ACCEPTED
3	Snoop request	HN to RN	SNP	Snp[*]Fwd	chi_snp_extension	Backward	BEGIN_REQ	END_REQ	TLM_UPDATED
								BEGIN_REQ	TLM_ACCEPTED
4	Snoop data response	RN to HN	WDAT	SnpRespData	chi_data_extension	Forward	BEGIN_PARTIAL_DATA BEGIN_DATA	END_PARTIAL_DATA END_DATA	TLM_UPDATED
								BEGIN_PARTIAL_DATA BEGIN_DATA	TLM_ACCEPTED
5	Snoop response	RN to HN	SRSP	SnpResp	chi_snp_extension	Forward	BEGIN_RESP	END_RESP	TLM_UPDATED
								BEGIN_RESP	TLM_ACCEPTED
6	Write data	RN to HN	WDAT	WriteCleanFull	chi_data_extension	Forward	BEGIN_PARTIAL_DATA	END_PARTIAL_DATA	TLM_UPDATED
								BEGIN_PARTIAL_DATA	TLM_ACCEPTED
7	Read data	HN to RN	RDAT	ReadClean	chi_data_extension	Backward	BEGIN_PARTIAL_DATA	END_PARTIAL_DATA	TLM_UPDATED
		SN to HN						BEGIN_PARTIAL_DATA	TLM_ACCEPTED
8	Read/write/data-less response	HN to RN	CRSP	CompAck	chi_ctrl_extension	Backward	ACK	ACK	TLM_UPDATED
9	Data-less transactions (Maintenance) i.e. Clean, Evict, Make etc request	RN to HN	REQ	Evict	chi_ctrl_extension	Forward	BEGIN_REQ	END_REQ	TLM_UPDATED
								BEGIN_RESP	TLM_ACCEPTED

- Note that when there is more than one data packet to be send the BEGIN_PARTIAL_DATA and END_PARTIAL_DATA is used. BEGIN_DATA and END_DATA phases mark the last data packet.
- If there is no delay between arrival of request BEGIN_REQ and acceptance of the request, then the phase can be updated to END_REQ while returning the call.
- If delay is there in arrival of request i.e. 'BEGIN_REQ' and acceptance of the same, then the non-blocking call can be returned immediately with 'TLM_ACCEPTED' as response. The request acceptor can then make another backward non-blocking call with phase set to 'END_REQ'.
- Similar approach, as BEGIN_REQ and END_REQ given above, can also be used for all the other non-blocking calls. Hence above table has 2 entries for each non-blocking call.

8 Implementation Guideline

The following sections describe an implementation of the specification. As such it is not part of the specification and may be subject to change in the course of implementation.

8.1 Payload Extension

This section is going to describe the extensions provided by the CHI TLM2.0 transactor package.

8.1.2 Structs for extensions

As outlined in section 3, there are six data structures representing the attributes of a CHI transaction. Along with the data members already described, they provide some utility functions to partially decode the signals and their meaning. These are:

```
struct common {
public:
    void reset();
    void set_txn_id(unsigned int);
    unsigned int get_txn_id() const;

    void set_src_id(unsigned int);
    unsigned int get_src_id() const;

    void set_qos(uint8_t qos);
    unsigned int get_qos() const;
};
```

The ‘common’ structure has fields common to all requests and response, namely txn_id, src_id and qos.

```
struct request {
    void set_tgt_id(uint8_t);
    uint8_t get_tgt_id() const;

    void set_lp_id(uint8_t);
    uint8_t get_lp_id() const;

    void set_return_txn_id(uint8_t);
    uint8_t get_return_txn_id() const;

    void set_stash_lp_id(uint8_t);
    uint8_t get_stash_lp_id() const;

    void set_size(uint8_t);
    uint8_t get_size() const;

    void set_max_flit(uint8_t data_id);
    uint8_t get_max_flit() const;

    void set_mem_attr(uint8_t);
    uint8_t get_mem_attr() const;

    void set_pcrd_type(uint8_t);
    uint8_t get_pcrd_type() const;

    void set_endian(bool);
    bool is_endian() const;

    void set_order(uint8_t);
    uint8_t get_order() const;

    void set_trace_tag(bool tg = true);
    bool is_trace_tag() const;
```

```
    void set_opcode(chi::req_optype_e op);
    chi::req_optype_e get_opcode() const;

    void set_return_n_id(uint16_t);
    uint16_t get_return_n_id() const;

    void set_stash_n_id(uint16_t);
    uint16_t get_stash_n_id() const;

    void set_stash_n_id_valid(bool = true);
    bool is_stash_n_id_valid() const;

    void set_stash_lp_id_valid(bool = true);
    bool is_stash_lp_id_valid() const;

    void set_non_secure(bool = true);
    bool is_non_secure() const;

    void set_exp_comp_ack(bool = true);
    bool is_exp_comp_ack() const;

    void set_allow_retry(bool = true);
    bool is_allow_retry() const;

    void set_snp_attr(bool = true);
    bool is_snp_attr() const;

    void set_excl(bool = true);
    bool is_excl() const;

    void set_snoop_me(bool = true);
    bool is_snoop_me() const;

    void set_likely_shared(bool = true);
    bool is_likely_shared() const;

    void set_rsvdc(uint32_t);
    uint32_t get_rsvdc() const; // Reserved for customer use.
};
```

The ‘request’ structure is used to capture signals/fields corresponding to transaction request started by RN and handled by HN, or requested by HN and completed by SN. These fields are as give in above table xxx.

```
struct snp_request {
    void set_fwd_txn_id(uint8_t);
    uint8_t get_fwd_txn_id() const;

    void set_stash_lp_id(uint8_t);
    uint8_t get_stash_lp_id() const;

    void set_stash_lp_id_valid(bool = true);
    bool is_stash_lp_id_valid() const;

    void set_vm_id_ext(uint8_t);
    uint8_t get_vm_id_ext() const;

    void set_opcode(snp_optype_e opcode);
    snp_optype_e get_opcode() const;

    void set_fwd_n_id(uint16_t);
    uint16_t get_fwd_n_id() const;

    void set_non_secure(bool = true); // NS bit
    bool is_non_secure() const;       // NS bit

    void set_do_not_goto_sd(bool = true);
```

```
    bool is_do_not_goto_sd() const;

    void set_do_not_data_pull(bool = true);
    bool is_do_not_data_pull() const;

    void set_ret_to_src(bool);
    bool is_ret_to_src() const;

    void set_trace_tag(bool = true);
    bool is_trace_tag() const;
};
```

The ‘snp_request’ structure is used to capture signals/fields corresponding to snoop transaction request started by HN and handled by RN, as give in above table xxx.

```
struct lcredit {
    lcredit() = default;
    void set_lcredits(int ncredits) { lcredits = ncredits; }
    void decrement_lcredits() { lcredits--; }
    unsigned get_lcredits() { return lcredits; }

private:
    int lcredits{0};
};
```

The struct to maintain the L-Credit information.

```
struct data {
    void set_db_id(uint8_t);
    uint8_t get_db_id() const;

    void set_opcode(dat_optype_e opcode);
    dat_optype_e get_opcode() const;

    void set_resp_err(uint8_t);
    uint8_t get_resp_err() const;

    void set_resp(dat_resptype_e);
    dat_resptype_e get_resp() const;

    void set_fwd_state(uint8_t);
    uint8_t get_fwd_state() const;

    void set_data_pull(uint8_t);
    uint8_t get_data_pull() const;

    void set_data_source(uint8_t);
    uint8_t get_data_source() const;

    void set_cc_id(uint8_t);
    uint8_t get_cc_id() const;

    void set_data_id(uint8_t);
    uint8_t get_data_id() const;

    void set_poison(uint8_t);
    uint8_t get_poison() const;

    void set_tgt_id(uint16_t);
    uint16_t get_tgt_id() const;

    void set_home_n_id(uint16_t);
    uint16_t get_home_n_id() const;
```

```
    void set_rsvdc(uint32_t);
    uint32_t get_rsvdc() const;

    void set_data_check(uint64_t);
    uint64_t get_data_check() const;

    void set_trace_tag(bool);
    bool is_trace_tag() const;
};
```

The data structure to be used in extension of payload for providing data in Request of Write & Read operations on WDAT and RDAT channels respectively or in Response of Snoop operation on WDAT channel.

```
struct response {
    void set_db_id(uint8_t);
    uint8_t get_db_id() const;

    void set_pcrd_type(uint8_t);
    uint8_t get_pcrd_type() const;

    void set_opcode(rsp_optype_e opcode);
    rsp_optype_e get_opcode() const;

    void set_resp_err(uint8_t);
    uint8_t get_resp_err() const;

    void set_resp(rsp_resptype_e);
    rsp_resptype_e get_resp() const;

    void set_fwd_state(uint8_t);
    uint8_t get_fwd_state() const;

    void set_data_pull(bool);
    bool get_data_pull() const;

    void set_tgt_id(uint16_t);
    uint16_t get_tgt_id() const;

    void set_trace_tag(bool);
    bool is_trace_tag() const;
};
```

The ‘response’ structure is used to capture signals/fields corresponding to all types of responses, namely

(a) transaction response on CRSP channel, sent by SN to HN or HN to RN, and
(b) snoop response on SRSP channel, sent by RN to HN

8.1.2 TLM extensions

The above structures are combined in corresponding payload extensions. There is one extension for each structure. This makes it modular, so that

- each Requester can create extension object as necessary and add to the payload, and make the blocking/non-blocking call for the transaction
- the receiver of the blocking/non-blocking call can use this extension to do the transaction.

For each of these structures’ ‘request’, ‘snp_request’, ‘data’ and ‘response’, the following extensions are defined.

TLM extension for ‘request’ and ‘response’:

```
struct chi_ctrl_extension : public tlm::tlm_extension<chi_ctrl_extension> {
```

```
    void set_txn_id(unsigned int id) { cmn.set_txn_id(id); }
    unsigned int get_txn_id() const { return cmn.get_txn_id(); }

    void set_src_id(unsigned int id) { cmn.set_src_id(id); }
    unsigned int get_src_id() const { return cmn.get_src_id(); }

    void set_qos(uint8_t qos) { cmn.set_qos(qos); }
    unsigned int get_qos() const { return cmn.get_qos(); }

    common cmn;
    request req;
    response resp;
};
```

TLM extension for 'snp_request' and according 'response'

```
struct chi_snp_extension : public tlm::tlm_extension<chi_snp_extension> {

    void set_txn_id(unsigned int id) { cmn.set_txn_id(id); }
    unsigned int get_txn_id() const { return cmn.get_txn_id(); }

    void set_src_id(unsigned int id) { cmn.set_src_id(id); }
    unsigned int get_src_id() const { return cmn.get_src_id(); }

    void set_qos(uint8_t qos) { cmn.set_qos(qos); }
    unsigned int get_qos() const { return cmn.get_qos(); }

    common cmn;
    snp_request req;
    response resp;
};
```

TLM extension for L-Credit flow control

```
struct chi_credit_extension : public
tlm::tlm_extension<chi_credit_extension>, public lcredit {
     chi_credit_extension() = default;
     tlm::tlm_extension_base* clone() const;
     void copy_from(tlm::tlm_extension_base const& ext);
};
```

TLM extension for 'data'

```
struct chi_data_extension : public tlm::tlm_extension<chi_data_extension> {

    void set_txn_id(unsigned int id) { cmn.set_txn_id(id); }
    unsigned int get_txn_id() const { return cmn.get_txn_id(); }

    void set_src_id(unsigned int id) { cmn.set_src_id(id); }
    unsigned int get_src_id() const { return cmn.get_src_id(); }

    void set_qos(uint8_t qos) { cmn.set_qos(qos); }
    unsigned int get_qos() const { return cmn.get_qos(); }

    common cmn{};
    data dat{};
};
```

Consideration for extensions

Instead of considering all the structures in one extension, as in case of AXI TLM, it is recommended to use one extension for each structure. Since, there are many combinations of data-based, snoop-based, snoop-less and data-less (maintenance) transaction, it is better to

make an extension for the current opcode, and send that extension with the payload.
Let us take as example a snoop based Read operation.

1. RN will add ‘chi_req_extension’ to payload and make the nb_transport_fw(..) call
2. In order to snoop another RN, the HN will add ‘chi_snp_req_extension’ to the payload. Now, payload will have two extensions.
3. After response from the snooped RN, HN will remove the ‘chi_snp_req_extension’ from the payload.
4. After getting response from HN, the initiator will process the response and remove the ‘chi_req_extension’ from the payload.

In such a scenario, if we use one extension having all the structures, since each of the structure has many fields, the extension structure will become very big and many of the fields may be un-used there. Also, since the src_id and tgt_id is common to each of the structure, for each hop of transaction, we need to retain these values.

8.2 Phases Declarations and Protocol traits

According to the specified protocol modeling 4 additional non-ignorable phases need to be defined:

```
// additional CHI phases
DECLARE_EXTENDED_PHASE(BEGIN_PARTIAL_DATA);
DECLARE_EXTENDED_PHASE(END_PARTIAL_DATA);
DECLARE_EXTENDED_PHASE(BEGIN_DATA);
DECLARE_EXTENDED_PHASE(END_DATA);
DECLARE_EXTENDED_PHASE(ACK);
```

Since these are non-ignorable, a specific protocol trait needs to be defined to comply with the TLM2.0 LRM:

```
using chi_payload = tlm::tlm_generic_payload;
using chi_phase = tlm::tlm_phase;
// chi protocol traits class
struct chi_protocol_types {
	typedef chi_payload tlm_payload_type;
	typedef chi_phase tlm_phase_type;
};
```

8.3 Socket Interfaces and Sockets

The standard TLM interfaces are re-used from the tlm base protocol specification:

Socket Interfaces

```
// the forward interface
template <typename TYPES = chi::chi_protocol_types>
using chi_fw_transport_if = tlm::tlm_fw_transport_if<TYPES>;

// The backward interface:
template <typename TYPES = chi::chi_protocol_types>
using chi_bw_transport_if = tlm::tlm_bw_transport_if<TYPES>;
```

Sockets

Based on the definitions so far, the initiator and target socket are declared as follows:

1. chi_initiator_socket

   This initiator socket is present on

   a) RN for doing transaction request to HN, and handling completer response from HN.

   b) HN for transaction request to SN

```
template <unsigned int BUSWIDTH = 32, typename TYPES = chi_protocol_types, int N = 1,
          sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
struct chi_initiator_socket
: public tlm::tlm_base_initiator_socket<BUSWIDTH, chi_fw_transport_if<TYPES>,
chi_bw_transport_if<TYPES>, N, POL> {
    //! base type alias
    using base_type = tlm::tlm_base_initiator_socket<BUSWIDTH, chi_fw_transport_if<TYPES>,
chi_bw_transport_if<TYPES>, N, POL>;
    /**
     * @brief default constructor using a generated instance name
     */
    chi_initiator_socket()
    : base_type() {}
    /**
     * @brief constructor with instance name
     * @param name
     */
    explicit chi_initiator_socket(const char* name)
    : base_type(name) {}
    /**
     * @brief get the kind of this sc_object
     * @return the kind string
     */
    const char* kind() const override { return "chi_trx_initiator_socket"; }
#if SYSTEMC_VERSION >= 20181013 // not the right version but we assume TLM is always bundled
with SystemC
    /**
     * @brief get the type of protocol
     * @return the kind typeid
     */
    sc_core::sc_type_index get_protocol_types() const override { return typeid(TYPES); }
#endif
};
```

2. chi_trx_target_socket

This socket inside Target is corresponding to the chi_initiator_socket

```
template <unsigned int BUSWIDTH = 32, typename TYPES = chi_protocol_types, int N = 1,
          sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
struct chi_target_socket : public tlm::tlm_base_target_socket<BUSWIDTH,
chi_fw_transport_if<TYPES>, chi_bw_transport_if<TYPES>, N, POL> {
    //! base type alias
    using base_type = tlm::tlm_base_target_socket<BUSWIDTH, chi_fw_transport_if<TYPES>,
chi_bw_transport_if<TYPES>, N, POL>;
    /**
     * @brief default constructor using a generated instance name
     */
    chi_target_socket()
    : base_type() {}
    /**
     * @brief constructor with instance name
     * @param name
     */
    explicit chi_target_socket(const char* name)
    : base_type(name) {}
    /**
     * @brief get the kind of this sc_object
     * @return the kind string
     */
    const char* kind() const override { return "chi_trx_target_socket"; }
#if SYSTEMC_VERSION >= 20181013
    /**
     * @brief get the type of protocol
     * @return the kind typeid
     */
    sc_core::sc_type_index get_protocol_types() const override { return typeid(TYPES); }
#endif
};
```