www.opencapi.org

OpenCAPI 3.1

Transaction Layer Specification

Version 1.0
28 January 2020

Approved

Approved for Distribution to OpenCAPI Members
Approved for Distribution to Non-Members for Learning Purposes Only

OpenCAPI 3.1 Transaction Layer Specification

OpenCAPI TL Specification Work Group
OpenCAPI Consortium

Version 1.0 (28 January 2020)

Printed in the United States of America February 5, 2021 (1.9.1.3p0.002).

All capitalized terms in the following text have the meanings assigned to them in the OpenCAPI Intellectual Property Rights Policy (the “OpenCAPI IPR Policy”). The full Policy may be found at the OpenCAPI Consortium website.

THE SPECIFICATION IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL LICENSOR, ITS MEMBERS OR ITS CONTRIBUTORS BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE SPECIFICATION.

Abstract

This document details the OpenCAPI TL specification. It is the work product of the OpenCAPI Consortium TL Specification Work Group.

This document is handled in compliance with the requirements outlined in the OpenCAPI Consortium Work Group (WG) process document. Comments, questions, etc. can be submitted to membership@opencapi.org.

Participants

Brian Allison, IBM, *Chair*

Michael Siegel, IBM, *Technical Editor*

Joe Breher, Western Digital Technologies, Inc

Harold Dozier, Micron Technology, Inc

Mark Fredrickson, IBM

Rick Hagen, NVIDIA Corporation

Paul Hartke, Xilinx, Inc

Curt Wollbrink, IBM

Contents

List of figures 7

List of tables 8

Revision log 10

About this document 11
Architecture compliance terminology 11
Conventions used in this specification 11
Bit and byte numbering 11
Representation of numbers 12
RTL notation 12
Notes 13
Engineering notes 13
Developer notes 13
Command flows and transaction diagrams 14
Command flow diagrams 14
Transaction diagrams 14

Terms 17

1. Overview 24
1.1 OpenCAPI protocol stack 25
1.2 Host operation modes 26
1.2.1 No attached device (C0, M0) 26
1.2.2 MEM-only mode (C0, M1) 26
1.2.3 Checkout mode (C1, M0) 27
1.2.4 Checkout with MEM (C1, M1) 27
1.3 Command ordering 27
1.4 Write fragmentation ordering and atomicity 27
1.4.1 Write fragmentation ordering and atomicity at the host 27
1.4.1.1 Partial write operations 27
1.4.1.2 64-,128-, 256-byte write operations 27
1.4.2 Write fragmentation ordering and atomicity at the AFU 28
1.4.2.1 Partial write operations 28
1.4.2.2 64-, 128-, 256-byte write operations 28
1.5 OpenCAPI device PA space specification 28
1.5.1 PA-to-RA mapping rules 29
1.6 Address translation 30
1.6.1 Effective to real address translation 30

2. TL and TLX command and response specifications 31
2.1 Handling multiple responses to a single command 35
2.1.1 TLX Read request getting multiple TL responses 35
2.1.2 TLX Write request getting multiple TL responses 36

2.1.3 TL read request getting multiple TLX responses. 36
2.1.4 TL write request getting multiple TLX responses 37
2.2 TL CAPP command packets 38
2.3 TLX AP command packets 50
2.4 TL CAPP response packets 71
2.5 TLX AP response packets 84

3. Virtual channel and data credit pool specification 92
3.1 Virtual channel 93
3.1.1 TLX command and response VC (TLX.vc) 93
3.1.2 TL command and response VC (TL.vc) 93
3.1.3 VC credit count specification 94
3.2 Data credit pool 94
3.2.1 TLX data DCP (TLX.dcp) 94
3.2.2 TL data DCP (TL.dcp) 95
3.2.3 DCP credit count specification 95
3.3 TL Virtual channel and service queues 96
3.3.1 Host TLX command handling 96
3.3.2 Host TLX response handling 98
3.4 Device TL virtual channel queues 98
3.5 Virtual channel dependency rules 99

4. The acTag table 101
4.1 acTag table contents 101
4.2 acTag table access 101
4.2.1 Error cases when accessing the acTag table 101
4.3 acTag entry management 101

5. DL frame format 103
5.1 DL frame control flit (64 bytes) 104
5.1.1 DL content 104
5.1.2 TL command/response content 105
5.1.3 Data transport, order, and alignment 105

6. TL and TLX template specifications 108
6.1 TLX receive and TL transmit template capability specification 110
6.2 TL receive and TLX transmit template capability specification 113
6.3 Control-flit rate capability 114
6.4 Metadata capability 115

7. Error detection 116
7.1 Error events 117

8. OpenCAPI profiles 124

Appendix A. AP (TLX) command transaction diagrams 131
A.1 AFU read with no intent to cache; 128 bytes 132

A.2 AFU DMA write; 128 bytes 133
A.3 AFU DMA partial write; 8 bytes 135

Appendix B. CAPP (TL) command transaction diagrams 136
B.1 CAPP memory read; 128 bytes 136
B.2 CAPP memory write; 128 bytes 137

List of figures

Figure 1. Big- and little-endian comparisons 12
Figure 2. Command flow example 14
Figure 3. Example TLX and TL transaction diagram 16
Figure 1-1. OpenCAPI stack 25
Figure 2-1. Address translation sequence: **xlate_touch** 67
Figure 3-1. TL command flow from the VC queue to the service queue 97
Figure 3-2. TLX command and response flow from the VC to the AFU protocol stack 99
Figure 3-3. VC dependency graph 100
Figure A-1. TLX and TL interaction: **rd_wnitc** 132
Figure A-2. TLX and TL interaction: **dma_w** 133
Figure A-3. TL and TLX interaction: **dma_pr_w** 135
Figure B-1. TL and TLX transaction: **rd_mem** 136
Figure B-2. TL and TLX transaction: **write_mem** 137

List of tables

Table 1. Architecture terms 11
Table 2-1. TL and TLX command operands 31
Table 2-2. The Resp_code specification for **xlate_done** 39
Table 2-3. The Resp_code specification for **intrp_rdy** 40
Table 2-4. The cmd_flag specification for **amo_rd** 56
Table 2-5. The cmd_flag specification for **amo_rw** 58
Table 2-6. The cmd_flag specification for **amo_w** 60
Table 2-7. The cmd_flag specification for **xlate_touch** (all forms) 65
Table 2-8. The Resp_code specification for **touch_resp** 72
Table 2-9. **touch_resp** Resp_code use by TLX command 73
Table 2-10. The Resp_code specification for **read_failed** 74
Table 2-11. **read_failed** Resp_code use by TLX command 76
Table 2-12. The Resp_code specification of **write_failed** 78
Table 2-13. **write_failed** Resp_code use by TLX command 79
Table 2-14. The Resp_code specification for **intrp_resp** 80
Table 2-15. **intrp_resp** Resp_code use by TLX command 81
Table 2-16. The Resp_code specification for **wake_host_resp** 82
Table 2-17. The Resp_code specification for **mem_rd_fail** 85
Table 2-18. **mem_rd_fail** Resp_code use by TL command 86
Table 2-19. The Resp_code specification for **mem_wr_fail** 88
Table 2-20. **mem_wr_fail** Resp_code use by TL command 90
Table 2-21. The Resp_code specification for **mem_cntl_done** 91
Table 3-1. VC maximum credit count specification 94
Table 3-2. DCP maximum credit count specification 95
Table 3-3. Summary VC and DCP assignments 95
Table 5-1. DL frame format showing CRC and “bad data flit” coverage 103
Table 5-2. DL frame loading to illustrate data ordering 107
Table 6-1. Template capability definitions 109
Table 6-2. Terms used in template capability specifications 109
Table 6-3. TLX receive/TL transmit template 110
Table 6-4. TL receive/TLX transmit template 113
Table 7-1. Error event specification 117
Table 8-1. Feature compliance requirement notation 124
Table 8-2. Profile specifications for TL commands 125
Table 8-3. Profile specifications for TLX commands 125
Table 8-4. Profile specifications for TL responses 126
Table 8-5. Profile specifications for TLX responses 126
Table 8-6. Profile specifications for TLX receive/TL transmit templates 127

Table 8-7. Profile specifications for TL receive/TLX transmit templates 128
Table 8-8. Profile specifications host operation modes 129
Table 8-9. Profile specifications supported page size 129
Table 8-10. Profile specifications supported dLength by TLX 129
Table 8-11. Profile specifications supported dLength by TL 129
Table 8-12. Profile specifications support of endianness data format by the TL 130

Revision log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was modified from the previous release of this document.

Revision date	Description
28 January 2020	Release of Approved OpenCAPI TL 3.1 specification.

About this document

This document provides the architectural specification of the OpenCAPI^{TM} transaction layer (TL and TLX).

Architecture compliance terminology

In architecture descriptions, certain terms carry meaning in addition to their normal use in English. The following terms are used within this architecture specification to describe the requirements an implementation must meet to be considered compliant.

Table 1. Architecture terms

Term	Description
invalid	Used for multi-bit fields where the contents are not reliable. The field or bus shall not be examined for any functional or error checking actions.
may	An architectural option indicating that an implementation is allowed to have this behavior or characteristic.
reserved	With respect to a field of a register or bus: • A reserved field shall be set to 0 by an implementation. • A reserved field shall not be examined by an implementation. With respect to a code point: • A reserved code point shall not be issued by a compliant implementation • A reserved code point shall cause a bounded undefined response (that is, it won’t hang the system). • A reserved code point may be used in future revisions of the architecture. The architecture may specify that the use of a reserved code point is an error condition.
shall	An architectural requirement indicating a required behavior or characteristic.
uncertain	Used for single-bit fields where the contents are not reliable. The field or bus shall not be examined for any functional or error checking actions.
undefined	When the value of a field or a bus is undefined, the value may vary between implementations and may vary for a particular implementation for different actions. An implementation shall not examine a field when its value is undefined for functional purposes. However, the field may be checked for errors in those cases where an implementation includes error checking (that is, parity, ECC and so on)

Conventions used in this specification

Bit and byte numbering

Throughout this document, little-endian notation is used, which means that bits and bytes are numbered in descending order from left to right.

Thus, in the description of a 4-byte field, bit 31 is the most significant bit (MSb) and bit 0 is the least significant bit (LSb). The corresponding byte numbering is also shown.

MSb																															LSb
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
byte 3								byte 2								byte 1								byte 0							

The big-endian and little-endian byte ordering are described in the *POWER ISA, version 3.0, Book I. Figure 1* compares big-endian and little-endian notation.

Figure 1. Big- and little-endian comparisons

LE	7	6	5	4	3	2	1	0
	Bit numbering within a byte							
BE	0	1	2	3	4	5	6	7

4-byte field with character data shown

LE	3	2	1	0
Content:	M	I	K	E
BE	0	1	2	3

Illustrating the difference between little endian and big endian storing to memory of the 4-byte field shown to the left.

Memory offset	LE stored	BE stored	
0	E	M	
1	K	I	
2	I	K	
3	M	E	

Representation of numbers

The notation for bit encoding is as follows:

- Hexadecimal values are preceded by an x and enclosed in single quotation marks. For example x‘0A00’. Bit numbering is little endian and, in this example, is 15 to 0.
- Binary values in sentences are shown in single quotation marks. For example ‘1010’. Bit numbering in is little endian and, in this example, is 3 to 0.
- ${}^{n}x$ means the replication of x, n times. That is, x is concatenated to itself n-1 times. ${}^{n}0$ and ${}^{n}1$ are special cases:
  - ${}^{n}0$ means a field of n bits with each bit equal to 0. For example, ${}^{5}0$ is equivalent to ‘00000’.
  - ${}^{n}1$ means a field of n bits with each bit equal to 1. For example, ${}^{5}1$ is equivalent to ‘11111’.

RTL notation

RTL notations are used to specify the architectural transformation performed by the execution of a command.

Notation	**Meaning**
←	Assignment.
||	Concatenation.
=, ≠	Equal, not equal relations.
≥, ≤	Greater than or equal to, less than or equal to relations.

Notation	Meaning
>, <	Greater than or less than relations.
+	Two’s complement addition.
-	Two’s complement subtraction, unary minus
$\vee$	Bitwise logical OR
$\wedge$	Bitwise logical AND
$\oplus$	Bitwise logical exclusive OR
Max(x,y)	Returns x when x ≥ y; otherwise returns y
Min(x,y)	Returns x when x ≤ y; otherwise returns y.
{x...y}	All integer values from x through y.
A = {x...y}	Returns true when A is a member of the set of integer values in the range of x through y.

Notes

This document contains engineering and developer notes.

Engineering notes

Engineering notes provide additional implementation details and recommendations not found elsewhere. The notes might include architectural compliance requirements. That is, the text might include *Architecture compliance terminology*. These notes should be read by all implementation and verification teams to ensure architectural compliance.

Engineering note

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin cursus hendrerit enim, vel tempus nibh ornare ut. Quisque ac augue eu augue convallis hendrerit. Mauris iaculis viverra ipsum nec dapibus. Nunc at porta libero. Curabitur luctus ultrices augue non pulvinar. Vestibulum mattis non ipsum at venenatis. Suspendisse euismod, neque et suscipit luctus, odio metus semper lectus, quis volutpat est libero quis nunc. Vivamus rutrum mauris sed tristique malesuada. Vivamus at augue vitae nisl cursus feugiat. Pellentesque efficitur sed nisi in dapibus. Curabitur vestibulum cursus arcu, ut mattis nisl.

Developer notes

Developer notes are used to document the reasoning and discussions that led to the current version of the architecture. These notes might also include recommended changes for future versions of the architecture, or warnings of approaches that have failed in the past. These notes should be read by verification teams and contributors to the architecture.

Developer note

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin cursus hendrerit enim, vel tempus nibh ornare ut. Quisque ac augue eu augue convallis hendrerit. Mauris iaculis viverra ipsum nec dapibus. Nunc at porta libero. Curabitur luctus ultrices augue non pulvinar. Vestibulum mattis non ipsum at venenatis. Suspendisse euismod, neque et suscipit luctus, odio metus semper lectus, quis volutpat est libero quis nunc. Vivamus rutrum mauris sed tristique malesuada. Vivamus at augue vitae nisl cursus feugiat. Pellentesque efficitur sed nisi in dapibus. Curabitur vestibulum cursus arcu, ut mattis nisl.

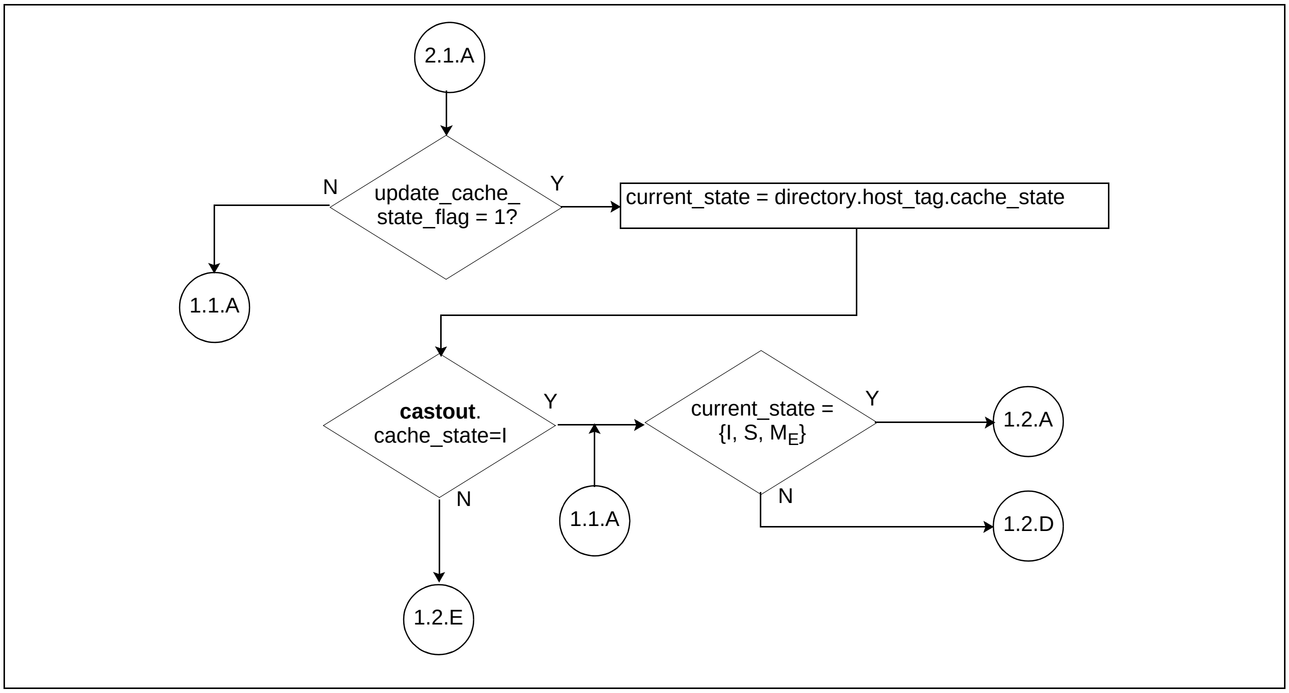
Command flows and transaction diagrams

Command flow diagrams

Command-flow diagrams show interactions within and across the different levels of the OpenCAPI protocol stack. Command flows use diamonds for decision blocks and rectangles for actions taken. Circles are used for on-page and off-page connectors and indicate a from-to direction based on the text content of the circle.

In *Figure 2*, a simple decision block with a state change and an off-page connector is shown. The text within the off-page connector has the format of “source page”.destination page”.”instance”. The off-page connectors shown in the figure is on page 1 of the figure[1] and is connecting to page 2 of the figure. On page 2, identical off-page connectors can be found. The instance indication allows for multiple connections to be shown between two pages. Connector 2.1.A illustrates a connection from page 2 to page 1 of *Figure 2*. An off-page connector can also be used to “connect” two spots on the same page as illustrated by connector 1.1.A. The direction of the arrow, into or out of a connector, decision block, or assignment-action block, indicates the direction of the sequence within the flow diagram.

Figure 2. Command flow example



Transaction diagrams

Transaction diagrams show the interaction between the TL and TLX layers and provide some illustrative notes for actions taken at the host protocol layer and the attached functional unit (AFU) protocol layer. In *Figure 3* on page 16, the diagram is broken into three vertical sections. From left to right, these are the AFU protocol layer notes, transactions between the TL and TLX layers, which are typically command and response packets, and the host protocol layer notes. Arrows indicate the direction in which the packet or action flows; for example, towards or away from the host (TL) layer.

[1] All multi-page figures contain a “page n of y” notation in the figure description.

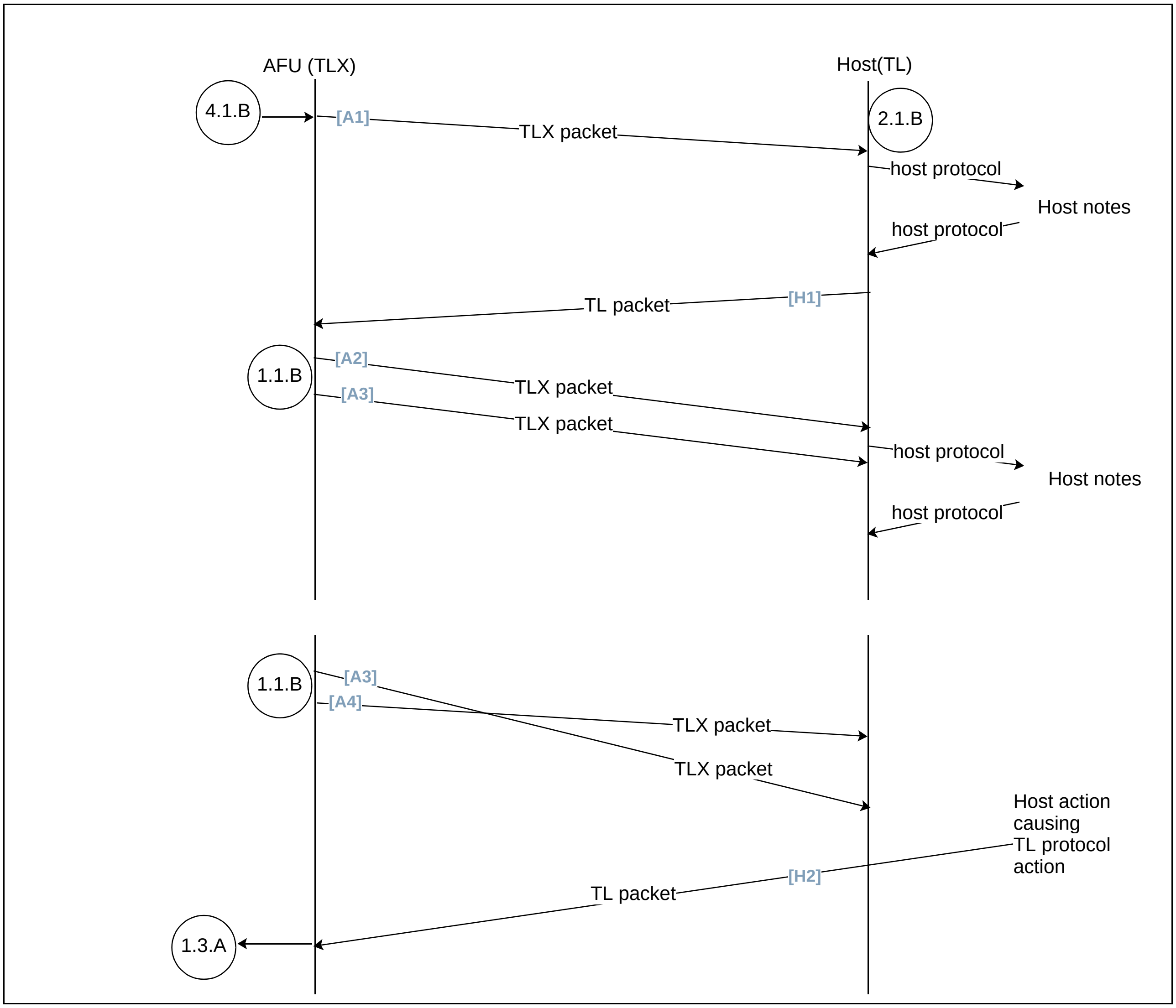
Circles are used for on-page and off-page connectors and indicate a from-to direction based on the text content of the circle. The text content is specified in the same manner for transaction diagrams as previously described for command flows. In addition to the specification of how connectors are used in command flows, in transaction diagrams, when a connector is used without an arrow, the transaction shown is one of multiple possible transaction outcomes. The use of this technique reduces the size of the transaction figure because the preceding set of transactions do not have to be repeated.

In *Figure 3*, connector 1.1.B illustrates an on-page connection without an arrow to indicate a different transaction out come. The prior events are assumed to have occurred when looking at the second instance of the 1.1.B connector. In the second case, one TLX packet has passed a previously issued TLX packet; this is something that can occur when two packets use different virtual channels. Connector 1.3.A shows an off-page connection to page 3, and connector 4.1.B shows an off-page connection from page 4.

Arrow numbering is included in transaction diagrams to simplify references to transactions. The form of the arrow references indicates the source of the transaction (AFU or Host) and the instance of the arrow. As seen in *Figure 3*, [A1] is the first arrow from the TLX packet transaction and [H1] is the first TL transaction.

A break in the vertical lines indicates where a new transaction illustration starts or ends.

Figure 3. Example TLX and TL transaction diagram



Terms

The following terms are used in this document.

{EA, address context}	A short hand notation to indicate an EA and *address context* pair. • Used when specifying an entry in an AFU L1 directory • Use in discussions about address translation from EA to either an RA or PA.
acLookUp(acTag)	This is a function call used in command flows and transaction diagrams. It converts an acTag found in a TLX command packet into the address context (ac) used by the host's platform architecture to authenticate and provide the function requested by the TLX command. The result of an acLookUp provides the error state of the address context provided. The state is shown as addressContext.state in the flows. The states are: 1. Good. The address context provided is valid. 2. Invalid acTag. The acTag entry in the acTag table is not valid, or the acTag is specified outside the acTag table range. See *Table 7-1* on page 117. 3. Invalid address context. The BDF and PASID associated with the acTag are invalid. The address context returned by the look up is not valid. See *Table 7-1* on page 117. The function description is host specific.
ACK	Acknowledgment.
address context	(ac or addressContext). Address context is the information associated with a particular BDF and PASID pair. The association is formed by actions specified by the host's platform architecture. For TLX commands, the acTag and the acTag table provide the BDF and PASID. See *Section 4 The acTag table* on page 101 for additional details.
address context space	A PASID paired with a BDF uniquely identifies the address space associated with a request. In OpenCAPI, a request is a TLX command.
address tenure	In a split transaction bus protocol, the commands and addresses are sent on the bus by the master before any data that might be associated with the transaction is moved. After the address tenure is completed, the status of the completion is examined. The data, if any is specified, is sent conditionally based on the status.
AFU	Attached functional unit. Architecturally, AFU refers to an end point unit or resource. Communication from the processor to the AFU goes through a protocol stack, transaction layer (TL), data link layer (DL), and physical medium layer (PHY). Command and data packets at the AFU interface are specified by the AFU command/data interface, which is the interface between the AFU protocol stack and the AFU.
AFU protocol	AFU protocol layer. This layer currently consists of: • AFU_C protocol layer • AFU_M protocol
AFU_C	A processing element that is able to generate and receive commands to obtain data in a checked-out (non-cached) state. It uses the AFU command/data interface to communicate with the AFU_C protocol stack. All addressing to the AFU_C protocol uses an EA only. It uses the AFU_C protocol stack to send and receive commands through the TLX. See *AFU type on page 26* for the different sub-types of an AFU_C.
AFU_C protocol	AFU_C protocol layer. This protocol specifies the sequences on the AFU command/data interface and the OpenCAPI packet interface (TLX boundary) for an AFU_C-defined AFU.
AFU_M	A processing element that receives commands to either provide or receive data. This element is a memory storage device and may be mapped to the system's memory address range. The attributes of the memory held by an AFU_M are managed by the operating system. It uses the AFU interface to communicate with the AFU_M protocol stack. All addressing to the AFU_M uses a PA only. See *AFU type on page 26* for the different subtypes of an AFU_M.
AFU_M protocol	AFU_M protocol layer. This protocol specifies the sequences on the AFU command/data interface and the OpenCAPI packet interface (TLX boundary) for an AFU_M-defined AFU.

alias	When address translation from one address type to another results in a many to one mapping, the set of addresses that map into the single address are referred to as alias of each other. During address translation, an alias is formed when two different addresses translate into the same address. For example: • Two or more physical addresses (PA) of an OpenCAPI device map to the same host real address (RA). • Two or more host RA map to a single attached OpenCAPI device's physical address.
AMO	Atomic memory operation. This operation performs an atomic update to a naturally aligned memory location. In some cases, this type of operation returns the original value of the memory location.
AP	Attached processor. Synonymous with AFU.
ATC	Address translation cache. The architecture describes a model for both a host ATC and an AFU ATC. See *Section 1.6 Address translation* on page 30.
BAR	Base Address Register.
back-off event	An event that causes a retry of an operation at some future time. The architecture specifies one type of back off event: long. The back off duration is controlled by a configuration space register specified in the OpenCAPI platform architecture.
BE	Byte enable.
CAPI	Coherent Accelerator Processor Interface.
CAPP	Coherent accelerator processor proxy.
command packet	TL/TLX construct. Contains command information for TL-to-TLX and TLX-to-TL communication.
convert2PA(RA)	This is a function call used in command flows and transaction diagrams. This coverts an RA seen on the host processor bus into a PA used by the attached OpenCAPI device. The mapping of an RA to a device PA is device and host platform dependent.
CRC	Cyclic redundancy check.
critical OW request	The following commands are provisioned to support a critical octword (OW) request: • **rd_wnitc** and all dot variants of this command • **rd_mem** A critical OW request is made when the address specified by the command is on a 32-byte (octword) boundary. Based on the size of the data block requested by the dLength specified, the address may not be naturally aligned. The requester is indicating that the OW specified by the command's address is latency critical. The requester is asking that the first data transfer associated with the first response for this command contain the critical OW. Responding with the critical OW first is optional.
data carrier	Data is transported between the TL and TLX in data carriers, which are defined as 64-byte data flits, or as 32- or 8-byte data fields found in control flits.
DCP	Data credit pool. Each command or response specified with immediate data consumes one or more data credits. To add a command or response to a DL frame's control flit, both the VC credit and the DCP credit must be atomically obtained. That is, you must have both to proceed to insert the command or response into the DL frame. Adding a command or response specified with immediate data to a DL control flit defines the order the data is sent towards its destination. See *Section 5.1.3 Data transport, order, and alignment* on page 105 for full details.
dError	Data error.
Device	The device refers to hardware and software attached via an OpenCAPI interface comprised of the PHYX, DLX, TLX Framer/Parser, TLX, AFU protocol stack, AFU protocol layer AFU interface and the AFU itself. See *Figure 1-1 OpenCAPI stack* on page 25.
DL	OpenCAPI data link layer found on the host processor.
dLength	Data length (dL).
DLX	OpenCAPI data link layer found on the external OpenCAPI device.

DMA	Direct memory access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.
dP, dPart	Data part (dP).
EA	Effective address. This is the address as seen by a program. Some host architectures refer to this as a virtual address (VA). Mapping from an EA to an RA requires address translation services.
ECC	Error correction code. A code appended to a data block that can detect and correct bit errors within the block.
Flit	An acronym for FLow control digITs. Typically used in networking to specify the smaller pieces that a larger network layer packet is broken into. See FLITs. In this architecture specification, a flit is associated with the specification of a DL frame and is defined as a 64-byte unit of data. Control and data flits are specified.
flit-cycle	The amount of time it takes 64-bytes to be either sent or received at the DL/TL or DLX/TLX interface.
host	The host refers to the host processor attached via an OpenCAPI link. It is comprised of the OpenCAPI PHY, DL, TL Framer/Parser, TL, the Host bus protocol stack interface and the hosts processors and other components that are implementation dependent on the host connected. See *Figure 1-1 OpenCAPI stack* on page 25.
host bus protocol layer	Specifies the sequences on both the host bus and at the host bus protocol layer and the OpenCAPI packet interface to: • Respond to snooped host bus commands from the OpenCAPI packet to the OpenCAPI transaction layer to initiate action at the target AFU. • Master commands on the host bus, per the specification found in the OpenCAPI packet, from the OpenCAPI transaction layer. Respond back to the source AFU at the conclusion of the host bus operation via an OpenCAPI packet to the TL layer.
immediate data	Data associated with a command or response. Immediate data is the data specified for a write operation (the command and the data travel in the same direction). A read response has immediate data (the response and the data travel in the same direction). A read command does not have immediate data; the data arrives with the response.
inbound	The direction from the attached OpenCAPI device towards the attached processor chip.
LRU	Least recently used. A policy for a caching algorithm that removes from the cache the item that has the longest elapsed time since its last access. An algorithm used to identify and make available the cache space that contains the data that was least recently used.
MEM	The memory-mapped owner of the line. The owner could be the memory controller or an the owner of a memory-mapped I/O space. Some coherency protocols refer to this as a point of coherency (POC).
metadata	Refers to information associated with a *naturally aligned data block*. This architecture specifies a 7-bit metadata field and a 72 bit extended-metadata field. Metadata is found in control flits where the template specifies the association of the metadata with the data. 7-bit metadata fields are found in templates x‘04’ through x‘09’. Extended metadata is found in templates x‘0A’ and x‘0B’.
minimum signed integer value	4-byte value: x‘8000_0000’ 8-byte value: x‘8000_0000_0000_0000’
MMIO	Memory-mapped input/output. Refers to the mapping of the address space required by an I/O device for Load or Store operations into the system’s address space.

mnemonic specification	The general format of a mnemonic for either commands or responses is based on a base command/response type and “dotted” subtypes. The following subtypes are currently specified: See the command specification to determine if a command is posted or non-posted. .be Byte enable field specified (dot-be). .d Data transfer (dot-d). Used only for **intrp_req** commands. .n Used for commands that require address translation (dot-n). If the address translation results in a miss in the ATC, the results of the address translation are used for the current operation, but are not loaded into the ATC. An implementation may: • Ignore this directive. • Store the results in a TLB. **Engineering Note** The dot-n form is expected to be used with a host implementation that has a multi-level ATC. This form of the command allows *warming up* the higher levels of the ATC hierarchy without installing into the more resource-precious level 1 ATC. .ow Octword data specified (dot-ow). Used for responses with immediate data consists of one or more control flits containing a 32-byte data field. The TL and TLX templates that support these control flit forms are specified in *Section 6 TL and TLX template specifications* on page 108. Responses with this form contain a dPart field with 32-byte address granularity. .xw X-word data specified (dot-xw). Used for responses with immediate data consists of one control flit containing an 8-byte data field. The TL and TLX templates that support these control flit forms are specified in *Section 6 TL and TLX template specifications* on page 108. **Developer note** The current set of TL/TLX templates limits the specification of dot-xw responses to 8-byte transfers. A future version could provide additional templates that support 16-byte data fields. The response encodes for the current set of dot-xw responses has bit 24 specified as ‘0’. To specify 16 bytes, bit 24 would be set to ‘1’.
MRU	Most recently used. One of the results of an *LRU* algorithm. The cache entry that has the shortest amount of elapsed time since its last access.
NACK	Negative acknowledgment.
naturally aligned data block	A data block containing L bytes is naturally aligned when the address specifying the location of the data block is an integer multiple of the length of the block. Where: i = {0, 1, 2, 3...} L = {64, 128, 256} A naturally aligned data block is located from byte address i * L through (i + 1) * L - 1. A command’s address specification may not be aligned as specified above. An unaligned address points to a naturally aligned data block of length L, with a starting address of $\text{adr}(63:(\log_2 L)) \vert\vert {}^{(\log_2 L)}0$.
nMMU	An abstraction of a host implementation-dependent construct that performs page-table walks based on the underlying page-table architecture as specified by the host architecture and host’s platform architecture. In the command flows and transaction diagrams, it returns: • nMMU_response.status = 0 when an address translation is successful. It returns page_size and access permissions. • nMMU_response.status <> 0 when software must be invoked to complete the requested address translation.
null control flit	A null control flit is defined as using template x‘00’. The 6-slot packet contains a 1-slot null command, and the remaining five slots are undefined. A return credit response found in slots 0 and 1 may be used to return credits. See *Section 6 TL and TLX template specifications* on page 108 for the specification of template x‘00’.

OMI	OpenCAPI memory interface. Additions to the OpenCAPI 3.0 TL specification: • Provide dot-ow and dot-xw formats of commands and responses • Provide TL and TLX template specifications x‘04’ through x‘08’ • Expand data carrier types to include 32- and 8-byte data fields found in some control flit formats • Specify metadata and extended metadata • Add critical OW specification for some read commands • Add *MAD* fields to some read commands
outbound	The direction from the processor chip to the attached OpenCAPI device.
PA	Physical address. This refers to the address space owned by an AFU$_M$ device. The host converts the RA to the AFU$_M$ device’s physical address space using configuration settings in the host that are determined during initialization of the attached OpenCAPI device. *A PA is not the result of address translation of an EA as might be the case in some host architectures. The host maps the device’s PA into its own (RA) address space.*
packet	TL/TLX unit of information. A command packet contains commands. A response packet contains response information. See the specification of command and response packets in *Section 2 TL and TLX command and response specifications* on page 31. Data is transferred in address-aligned: • 64-byte data flits • 8-byte data fields specified in some template specifications found in *Section 6* • 32-byte data fields specified in some template specifications found in *Section 6*
PHY	The PHY layer interfaces to the DL and the network. This is the bit stream level specifying the electrical and optical transmission medium as well as the network interconnect topology. The current specification for the network is a point-to-point connection.
PHYX	On the OpenCAPI device, the PHYX layer interfaces to the DLX and the network. This is the bit stream level that specifies the electrical and optical transmission medium as well as the network interconnect topology. The current specification for the network is a point-to-point connection.
pL, pLength	Partial length.
POC	Point of coherency. See definition of *MEM*.
RA	Real address. A real address is the result of address translation of an EA. Some host architectures refer to this as a physical address; this specification reserves the term physical address for other purposes. See the definition of *PA*.
Reserved/R	Indicates that a field or bit specification is reserved. A reserved field is set to zero and shall not be examined by an implementation. See *Architecture compliance terminology* on page 11.
response packet	TL construct that contains response information to commands. Used for TL-to-TLX and TLX-to-TL communication.
responder	TL or TLX that accepts a command, services the command, and sends back a response TL/TLX packet that provides data, when required, and status of the service to the command.
requester	TL or TLX that issues a command. The requester collects all responses returned by the responder, if any, to determine the status of the service provided by the command. When the command is posted, responses are not returned.
RTL	Register transfer language.
segment	When used in reference to data, a segment refers to a naturally aligned 64-byte portion of a data transfer. For example, a 256-byte data transfer contains four segments.

service queue	The members of a service queue are an ordered set of commands. The commands are selected by applying a hash against the VC, BDF, PASID and stream_id associated with the command.The hash results in the selection of a specific service queue. The hash is both implementation and command dependent. The hash is command dependent because not all commands are specified with a BDF, PASID and stream_id. Commands that are not specified with a VC do not enter a service queue. Per VC, the following operands may be included in the hash: TLX.vc.0 This VC is used for most responses, and the hash is the VC. TLX.vc.3 Contains various read and write TLX commands as well as **assign_actag**. The **assign_actag** command is serviced before entering into a service queue. All other commands are sorted using a hash based on the VC, BDF, PASID and stream_id. When the hash specified is used for a VC, the hash is perfect and the resulting service queue is identical to the definition of a *virtual queue*. When an implementation removes hash terms from the VC-specific specification, the hash is not perfect. There is at least one service queue per VC supported by the implementation.
slot	A slot is a 28-bit granule used to specify a TL or TLX command or response packet.
SUE	Special uncorrectable error. Refers to error detection and attempted correction to a block of data. A SUE indicates that an error was detected upstream from the present error detection logic. The use of SUE indications aids in determining error origination as part of a first error incident reporting scheme.
TL	OpenCAPI transaction layer found on the host processor. • Interfaces to the DL and the protocol layer. Responsible for command-packet formation and response-packet handling and formation. Ensures that the order of data sent to the DL matches the command- and response-packet order sent to the DL. • Manages data flits, 8- and 32-byte data carriers specified in some control flits from the DL. Associates the data with the command or response packet that was received prior to the arrival of the data. The command- and response-packets contain data descriptors that enable this association. • Performs flow control. • Performs error handling and control. • Manages all *service queues* associated with each virtual channel. Order is retained within virtual channels.
TLB	Translation lookaside buffer. An on-chip cache that holds the translation of an effective address (EA) to a real address (RA). A TLB caches page-table entries for the most recently accessed pages, thereby eliminating the necessity to access the page table from memory during load-store operations.
TLX	OpenCAPI transaction layer found on the external OpenCAPI device. • Interfaces to the DLX and the protocol layer. Responsible for command packet formation and response packet handling and formation. Ensures that the order of data sent to the DLX matches the command and response packet order sent to the DLX. • Manages data flits, 8-, and 32-byte data carried in some control flits from the DLX and associates the data with the command or response packet that was received prior to the arrival of the data. The command and response packets contain data descriptors that enable this association. • Flow control. • Error handling and control.
UE	Uncorrectable error. Refers to error detection and attempted correction to a block of data. An uncorrectable error indicates that an error was detected and the attempted correction failed.
VC	Virtual channel. See *Section 3 Virtual channel and data credit pool specification* on page 92, and the specification of all commands and responses in this section.
virtual queue	The specification of a *service queue* describes the VC-specific hash required to form a service queue from a virtual queue. The members of a virtual queue are an ordered set of commands received from a VC. That is, the ordering of the commands found in the VC shall be retained when adding commands from the VC to a virtual queue.
warming up	The process of loading or populating a cache with a set of valid data.
write class command	A command that is used to write data to a destination. The source of a write class command is also the source of the data.

xlate_result = adr_xlate(EA, addressContext)	This is a function call used in command flows and transaction diagrams. This returns the host's results from an address translation. The function returns an RA (xlate_result.RA) and a status (xlate_result.status). The status returned is: 1. Complete. Address translation completed successfully with an RA provided. The ATC may have been updated with the result. 2. rty_req. Indicates that the address translation could not be completed at this time. The operation may be attempted at a later time. 3. xlate_pending. Indicates that the address translation could not be completed. The ATC did not contain the translation and software was invoked. An asynchronous **xlate_done** TL command is sent when the software actions have completed. **Engineering note** The Resp_code=xlate_pending is sent in a **read_failed**, **touch_resp**, or **write_failed** response packets. These TL responses shall precede the **xlate_done** command in the TL.vc.0 virtual channel.

1. Overview

The OpenCAPI transaction layer specifies the control and response packets passed between a host and an OpenCAPI device. The transaction layer implemented on the host is referred to as the TL. The transaction layer implemented on the OpenCAPI device is referred to as the TLX.

On the host, the transaction layer converts:

- Host-specific protocol requests into transaction-layer-defined commands.
- TLX commands into host-specific protocol requests. When the host protocol completes, it provides responses to the TLX commands when required.
- TLX responses into responses for host-initiated requests.

On the OpenCAPI device, the transaction layer converts:

- AFU-specific protocol requests into transaction-layer-defined commands.
- TL commands into AFU-specific protocol requests. When the AFU protocol completes, it provides responses to the TL commands when required.
- TL responses into responses for AFU-initiated requests.

Working together, the TL and TLX provide a standard method to bridge between a host protocol architecture and an AFU protocol architecture. This is accomplished by the exchange of command and response packets specified by the OpenCAPI transaction layer specification.

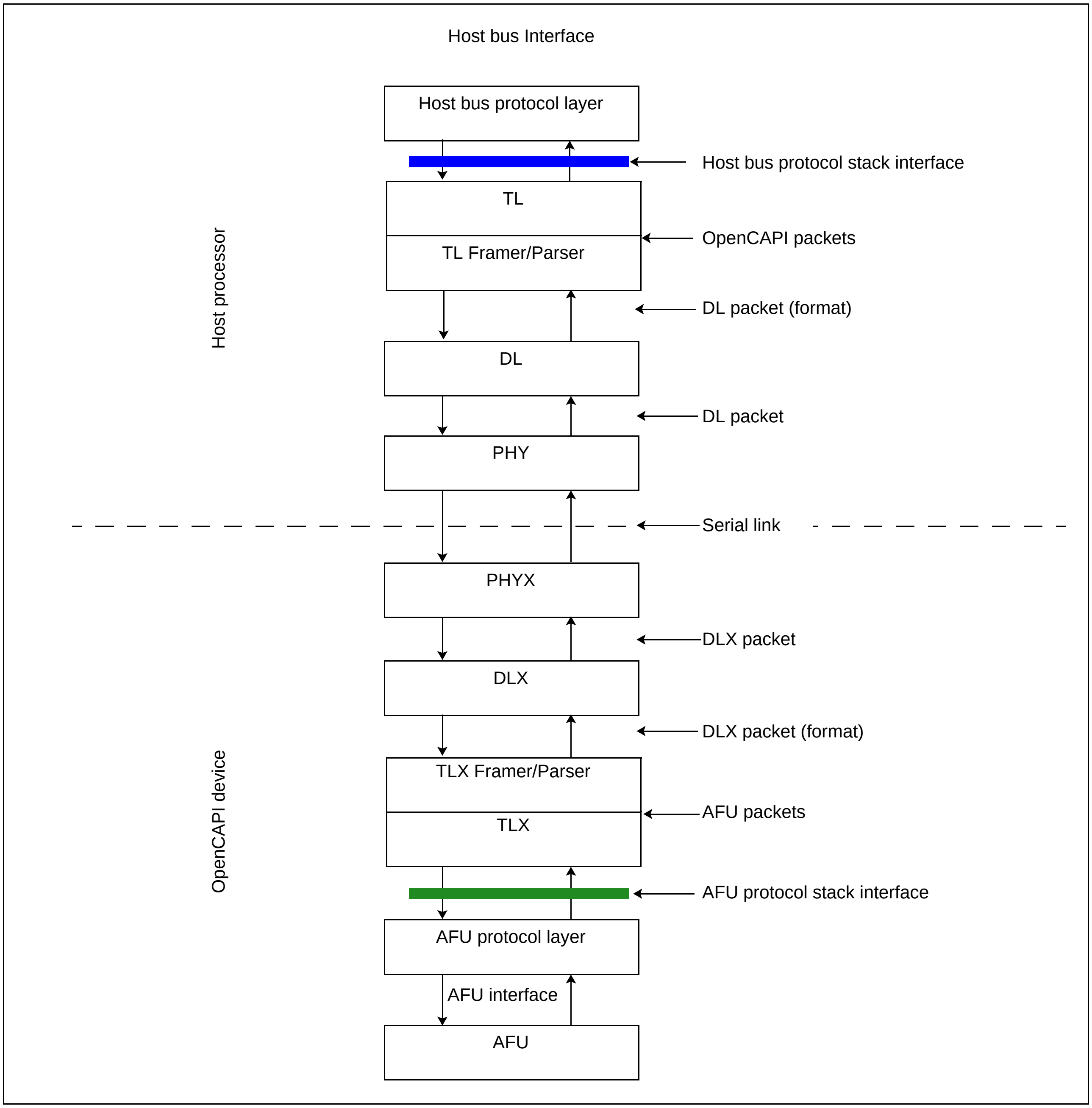
Version 3.1 of this specification builds on the version 3.0 base and adds OpenCAPI Memory Interface (OMI) extensions. These extensions:

- Provide dot-ow and dot-xw formats of commands and responses
- Provide TL and TLX template specifications x‘04’ through x‘08’
- Expand data carrier types to include 32- and 8-byte data fields found in some control flit formats
- Specify metadata and extended metadata
- Add critical OW specification for some read commands
- Add *MAD* fields to some read commands

1.1 OpenCAPI protocol stack

Figure 1-1 on page 25 shows the OpenCAPI protocol layers.

Figure 1-1. OpenCAPI stack



1.2 Host operation modes

The interface between the host and the AFU can be implemented with varying levels of complexity. Interoperability with an AFU implementation is dependent on the operation mode supported by the host and the requirements of the AFU.

The various combinations of AFU capabilities are broken into two subclasses, ***AFU_C*** and *AFU_M*, and each subclass is broken into two types.

AFU type	Description
AFU_{C0}	(C0 or none). There is no visible-to-the-host processing element. The host never sees any commands sourced by the TLX. While a processor element might not be visible to the host, it may still be present. If it is present, it shall not cache any lines in any coherent data valid state and shall not rely on the host's coherency protocol for correct operation.
AFU_{C1}	(C1 or type 1 processing element). A processing element with no cache. An AFU_{C1} may issue TLX commands to the host. It uses an EA to access host system memory. The host provides address translation and access to system memory.
AFU_{M0}	(M0 or none). There is no host system address space mapped to this device. That is, host system address space shall not be mapped to this device. Configuration space may be specified for this device.
AFU_{M1}	(M1 or type 1 MEM). A range of host system address space shall be assigned to this device. This address range shall be accessible only through the host (TL-to-TLX interactions). The host shall use the PA to access data. **Engineering Note** The address range assigned to this type of device may be limited to MMIO space only or may include memory that is manged by the operating system; for example, memory that is backed by DRAM and that can be migrated to disk as needed.

The following sections describe the combinations of AFU_C and AFU_M devices on a single OpenCAPI device.

1.2.1 No attached device (C0, M0)

No device is attached to the OpenCAPI interface. No transactions occur.

1.2.2 MEM-only mode (C0, M1)

In this mode of operation, the AFU appears to be a memory controller with an address space mapped into the host's system address space. Access by the host uses the PA. See the specification of an *AFU_M*.

Developer note

While a processor element might not be visible (C0), it might still be present. Examples of this type of configuration that meet the above requirements are:

- An encrypted memory device. In this device, the data is encrypted/decrypted when the data is written to the M1 or read from the M1. The processing element that performs the encryption/decryption is not visible to the host. Topologically, the processing element is between the memory and the host.
- A memory cache device. The cache is managed by a processing element. The cache exists to reduce latency, and the cache states are not related to the host's coherency protocol. Data might be fetched or stored into the cache. The host cannot tell this is happening except for an improvement in performance.

1.2.3 Checkout mode (C1, M0)

In this mode of operation, the AFU appears as a processing element without a cache. It may have a non-coherent scratch pad memory, which is used for local processing only. Access to system memory is permitted as coherent, no-intent-to-cache actions. The AFU shall use an EA for these requests. The host shall perform address translation to enable access to system memory.

1.2.4 Checkout with MEM (C1, M1)

In this mode of operation, the AFU appears as a processing element without a cache. It may have a non-coherent scratch pad memory, which is used for local processing only. Access to system memory is permitted as coherent, no-intent-to-cache actions. The AFU shall use an EA for these requests. The host shall perform address translation to enable access to system memory.

In addition, the AFU provides a memory controller function with an address space mapped into the host’s system address space. Access by the host uses a PA. See the specification of an AFU_M.

1.3 Command ordering

Ordering within a VC is maintained through the TL/TLX, but it is not assured after the command has moved to the upper protocol layers (host and AFU) as described in *Section 3 Virtual channel and data credit pool specification* on page 92.

1.4 Write fragmentation ordering and atomicity

1.4.1 Write fragmentation ordering and atomicity at the host

Write commands issued by the AFU may be fragmented by the host. The following sections specify the atomicity of the fragments and the order in which the updates become globally visible.

1.4.1.1 Partial write operations

These are TLX commands found in the following command classifications: pr_dma_write, atomics.r, atomics.rw, and atomics.w.

Minimum guaranteed write atomicity is specified as 16 bytes when aligned on a 16-byte address boundary. When the partial write operation is not specified with a naturally aligned address, atomicity may be reduced to a single byte. Data shall be globally visible in increasing address order.

1.4.1.2 64-,128-, 256-byte write operations

These are TLX commands restricted to 64-, 128-, or 256-byte naturally aligned write operations. These are TLX commands found in the following command classification: dma_write.

Minimum guaranteed write atomicity is specified as 64 bytes. When the write operation specifies 128 or 256 bytes and the host fragments the write operation, there is no ordering guarantee for the data segments written.

Developer note

Applications are expected to use 64-byte or smaller writes for synchronizing required events; for example, when writing a semaphore. Larger data transfer sizes are expected to be used for data transfer efficiency.

1.4.2 Write fragmentation ordering and atomicity at the AFU

Write commands issued by the host may be fragmented by the AFU. The following sections specify the atomicity of the fragments and the order in which the updates become globally visible.

1.4.2.1 Partial write operations

These are TL commands found in the following command classifications: pr_mem_write and configuration.

Minimum guaranteed write atomicity is specified as 16 bytes when aligned on a 16-byte address boundary. When the partial write operation is not specified with a naturally aligned address, atomicity may be reduced to a single byte. Data shall be globally visible in increasing address order.

Developer note

The architecture does not currently provide TL commands that are able to specify partial write operations where the data and address are not naturally aligned.

1.4.2.2 64-, 128-, 256-byte write operations

These are TL commands restricted to 64-, 128-, or 256-byte naturally aligned write operations. These are TL commands found in the following command classification: mem_write.

Minimum guaranteed write atomicity is specified as 64 bytes. When the write operation specifies 128 or 256 bytes and the AFU fragments the write operation, there is no ordering guarantee for the data segments written.

1.5 OpenCAPI device PA space specification

An OpenCAPI device may have the following three PA spaces specified:

1. Configuration space shall be specified for the device.
2. System memory space may be specified for the device.
3. MMIO space may be specified for the device.

Engineering note

MMIO space notes: Access to MMIO should be non-blocking. Retrying access to MMIO space is not prohibited by the architecture.

The configuration space is accessed by using the **config_read** or **config_write** commands. The PA specified for this space is separate from the system memory space and the MMIO space. The host may:

- Provide a configuration address BAR to access this space using a direct access load/store model.

- Provide an MMIO register set to access this space using an indirect access method.

This architecture does not specify the application of *metadata* to a device's configuration space.

The system memory space is memory space owned by the OpenCAPI device that is mapped to the host's system memory. The PA for system memory space is defined to start at offset 0. The host differentiates between the different system memory spaces of different OpenCAPI devices by providing a configuration address BAR for each attached device.

The MMIO space shares the PA space used by the system memory space. It is specified by a fixed offset from PA 0 which is specified in the OpenCAPI device's configuration space. The host differentiates the MMIO spaces of different OpenCAPI devices by providing a configuration address BAR for each attached device. Access to MMIO space is sensitive to the operand length and the command specified. The device literature should provide information on how to correctly access MMIO space.

- A device may not support all operand lengths provided by the architecture when accessing a specific address found in MMIO space. If an MMIO access does not use a correct operand size for the address specified, an unsupported-operand-length Resp_code shall result.
- A device may not support all commands provided by the architecture when accessing a specific address found in MMIO space. If an MMIO access does not use a correct command for the address specified, a Failed Resp_code shall result.
- All accesses to MMIO space shall result in a single response from the device. That is, when a dLength of 128 or 256 bytes is permitted, the device shall respond with the same dLength used in the command.

System and MMIO spaces are expected to be contiguous based on the configured starting PA and size. Access to unimplemented addresses results in the following:

- Read access to an unimplemented PA shall return all 1s data.
- Write access to an unimplemented PA shall result in discarded data.

1.5.1 PA-to-RA mapping rules

Real addresses (RA) are mapped into the physical address (PA) space specified for a device. This eliminates any requirement placed on the OpenCAPI device to have knowledge of the host's real address space or how the OpenCAPI device's PA space is mapped into it. The following rules place restrictions on the OpenCAPI device's specification of its PA space.

1. No address aliasing for PA-to-RA translation. That is, a PA for any specific device attached to an OpenCAPI link (PA + unique interface) translates into a unique RA. The address translation is specified by address ranges configured by software.
2. No address aliasing for RA-to-PA translation. The host protocol is provided with a single POC for each RA.

1.6 Address translation

1.6.1 Effective to real address translation

Effective to real address translation is specified by the host's architecture. The TL architecture model assumes a host address translation cache (ATC) that holds valid effective (EA) to real address (RA) translations. The ATC contains, at a minimum, a valid indication, the page size, the page size aligned starting effective address (EA), the address context of the translation in the form of the BDF and PASID, page write permission (W) and the host's corresponding real address.

The architecture supports multiple page sizes. See the specification of $\log_{2\_page\_size}$ on page 33 and page size capability recommendations found in *Table 8-9 Profile specifications supported page size* on page 129.

An implementation may choose to provide additional fields, or may replace some of the fields listed above with other host specific content. Since the architecture assumes that the contents of an ATC entry contain the fields specified by the TL architectural model, any implementation specific alterations shall be done in such a manner that the differences are not externally observable.

The architecture model does not require, but allows for, a multi-level ATC. Higher level ATC might have a smaller capacity and have faster access than a lower level ATC that have more capacity and longer access latency. The structure of a host's ATC is outside the scope of this architecture.

The TL architectural model assumes that TLX commands with an EA specified go through effective to real address translation before execution on the host protocol bus. See *Section 3.3 TL Virtual channel and service queues* on page 96 for additional details.

An AFU can warm up the host's ATC by using the TLX **xlate_touch** command. See the command description for additional details.

2. TL and TLX command and response specifications

This section specifies all command and response types originated in the TL and the TLX. Commands originating in the TL are referred to as CAPP command packets (CAPP_cmd). Commands originating in the TLX are referred to as AP command packets (AP_cmd). Responses originating in the TL are referred to as CAPP response packets (CAPP_response). Responses originating in the TLX are referred to as AP response packets (AP_response)

In the subsections of this chapter descriptions use the following format:

Command descriptive name	**mnemonic**	Assigned opcode
command classification	VC used, DCP used (immediate data)	28-bit slot count

Table 2-1 lists the command operands used in the TL and TLX command and response specifications. See *Terms* on page 17 for definitions of terms used in these specifications.

Table 2-1. TL and TLX command operands (Page 1 of 5)

Operand mnemonic	Field width	Description
acTag	12	Address context tag. The address context tag is managed by the AFU. The acTag is used as an index into a host table that contains the BDF and PASID associated with the acTag. The OpenCAPI device learns its Bus number during a **config_write,** T=0 operations. The function and device numbers are assigned by the attached OpenCAPI device's implementation and cannot be modified by any configuration actions. The OpenCAPI device shall be assigned at least one PASID, and may be assigned more than one PASID, by host software during the initialization and operation of the device. The BDF and PASID are used for address translation authorization and operation validation.
AFUTag	16	Unique handle specifying the AFU and command instance. Provided by the AFU that is requesting command services of the TLX. A TL response to a single TLX command may be broken into multiple TL response packets. When this occurs, all responses associated with the TLX command shall return the same AFU Tag value. **Engineering Note** The TL shall not use the AFUTag for any purpose other than as data to complete the contents of a response packet, or when forming an **xlate_done** or **intrp_rdy** TL command packet. Any retirement rules specified by a device implementation for the AFUTag shall not be checked by the TL. **AFU tag retirement recommendations:** • For non-posted commands, the AFU should not reuse an AFUTag until all responses for the command have been received.
BDF	16	Bus device function. This is the identifier of a TLX requester. See acTag *on page 31* for additional details.
Byte enable	64	(BE) This field is found in commands with dot-be mnemonic specifications. Valid only for write class commands.

Table 2-1. TL and TLX command operands (Page 2 of 5)

Operand mnemonic	Field width	Description
CAPPTag	16	Unique handle specifying the host CAPP and command instance. Provided by the CAPP that is requesting command services of the TL. **Engineering Note** The TLX shall not use the CAPPTag for any purpose other than as data to complete the contents of a response packet. Any retirement rules specified by the host implementation for the CAPPTag shall not be checked by the TLX. **CAPPTag retirement recommendations:** • For non-posted commands, the CAPP should not reuse a CAPPTag until all responses for the command have been received.
cmd_flag	4	Specifies execution behavior for commands and responses specified with this field. The command or response specification includes the behavior specification for the cmd_flag when the field is specified.
cmd_opcode	8	Specifies the operation to be performed.
credit_return	48	Specifies the number of credits returned to the VC and DCP credit pools. The credits are returned in fixed subfield locations in a 2-slot (56-bit) TL or TLX response packet. See the specification for **return_tlx_credits** and **return_tl_credits** for the format of the field. Each VC credit allows for a single command or response to be sent in the virtual channel. Each DCP credit allows the sending of one *data carrier*.
dLength	2	Data length (dL). Indicates the number of data bytes associated with a command or response packet. This 2-bit field indicates a length of: 00 32 bytes when the command is **pad_mem** or when in response to a **pad_mem** command is **mem_wr_response** or **mem_wr_fail**. Reserved for all other commands and responses. 01 64 bytes. This field value shall be used in a response packet when the command is a partial read or write operation. 10 128 bytes. Reserved when the command is a partial read or write operation. 11 256 bytes. Reserved when the command is a partial read or write operation. When the dLength field in the response packet does not match the full amount of data requested by the command, the dPart field is used to indicate the offset within the *naturally aligned data block* specified by the command's address. For example, in the multiple responses to a single TLX read command, the AFUTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, continuing with the above example, the TLX may see the values of dPart returned in any order. Support for 256 bytes is optional for the TLX and AFU. See *Table 8-10 Profile specifications supported dLength by TLX* on page 129.

Table 2-1. TL and TLX command operands (Page 3 of 5)

Operand mnemonic	Field width	Description
dPart(1:0)	2	Data part (dP(1:0) or dPart(1:0)). Indicates the data content of the current response packet. Read requests can be 64, 128, or 256 bytes in length. This field indicates the starting offset from the naturally aligned data block specified by the address provided in the read command. The amount of data transfered due to this response packet is found in the dLength field. 00 Offset at 0 bytes. This field value shall be used for response packets when the command is a partial read or write operation. 01 Offset at 64 bytes. This field value *shall not* be used when the dLength specifies 128 or 256 bytes. Reserved when the command is a partial read or write. 10 Offset at 128 bytes. This field value *shall not* be used when the dLength specifies 256 bytes. Reserved when the command is a partial read or write. 11 Offset at 192 bytes. This field value *shall not* be used when the dLength specifies 128 or 256 bytes. Reserved when the command is a partial read or write. The presence of this field in a command allows for multiple responses to be returned for a command. For example, a 256-byte read command such as **rd_wnitc** may result in four responses with the dPart field taking on all four states. **Developer note** The constraints placed on dPart are to ensure that responses specify naturally aligned data blocks. This is intended to simplify the design and the verification state space.
dPart(2:0)	3	Data part (dP(2:0) or dPart(2:0). Indicates the data content of the current response packet. This field indicates the starting offset from the naturally aligned data block specified by the address provided in the read command.This extended version of dPart is used for dot-ow variants of **mem_rd_response** and **read_response**. The extended field width allows the specification of offsets with 32-byte granularity. The data is sent in 32-byte *data carriers*. For example, a 256-byte read command such as **rd_wnitc** results in eight responses with the dPart(2:0) field taking all eight states. A **pr_rd_wnitc** uses a single response and is restricted to an offset of 0 bytes. 000 Offset at 0 bytes. 001 Offset at 32 bytes. Reserved when the command is a partial read or write. 010 Offset at 64 bytes. Reserved when the command is a partial read or write. 011 Offset at 96 bytes. Reserved when the command is a partial read or write. 100 Offset at 128 bytes. Reserved when the command is a partial read or write. 101 Offset at 160 bytes. Reserved when the command is a partial read or write. 110 Offset at 192 bytes. Reserved when the command is a partial read or write. 111 Offset at 224 bytes. Reserved when the command is a partial read or write.
E	1	Operand endianness. Used for mem_atomics.* and atomics.* class commands to specify the endianness of the operands. For bitwise logical operations, the endianness of the operands does not change the result. The field is specified as follows: 0 Operands are little endian. 1 Operands are big endian.
EA	52, 59, 64	Effective address (also referred to as the VA or virtual address by some host architectures). Length specification is dependent on the command issued and is noted in the command specification.
$\log_2$_page_size	6	Log_2 value of the page size determined by the host when executing **xlate_touch**. The value of the page size touched in bytes is specified as $2^{\text{bin2dec}(\log_2\text{\_page\_size})}$. See *Table 8-9 Profile specifications supported page size* on page 129. Values corresponding to a page size of 1-byte to 2K-bytes are reserved.
MAD	8	Memory access directive. Specifies host directives when accessing OpenCAPI memory devices (AFU_M). The specification of this field is found in the host's platform architecture. The AFU_M may ignore this field.

Table 2-1. TL and TLX command operands (Page 4 of 5)

Operand mnemonic	Field width	Description
Meta	7	Metadata. This 7-bit field specifies the metadata for a data block held in memory. The size of the data block is implementation dependent. The TL architecture specifications provides for 7 bits of metadata for 8-, 32- and 64-byte data blocks. The specification of the metadata is outside the scope of this architecture and is found in the host and OpenCAPI device's documentation. An implementation shall transform the metadata, if necessary, when 8- and 32-byte naturally aligned data blocks are aggregated into 64-byte naturally aligned data blocks.
Object_handle	64/68	Used by message class commands. For TL commands, the object handle is specified by the OpenCAPI device manufacturer and is loaded into a table maintained by the device software. It is accessed by the host based on a method specified by the host's OpenCAPI platform architecture. For TLX commands, the object handle is specified by the host architecture and is loaded into a table held in the device's MMIO space. The OpenCAPI device manufacturer specifies the location of the MMIO space, and it is provided to the host through the device software.
PA	58, 59, 64	Physical address. Translation from the host RA to the AFU$_M$ PA is performed by the host and configured during device initialization. The AFU's configuration space provides the information about the topology of the physical address space held by the OpenCAPI device. Types of address spaces are: • Address space shared with the system. This excludes MMIO space. • MMIO space. • Configuration address space. This address space may be directly memory mapped and use a simple load/store model, or it may be accessed using indirect address methods. The choice is host dependent and is transparent to the OpenCAPI device and TL protocol.
PASID	20	This term identifies the user process associated with a request. In OpenCAPI, a request is a TLX command. See acTag *on page 31* for additional details.
pLength	3	(pL) Partial length. Specifies the number of data bytes specified for a partial write command. The address specified shall be naturally aligned based on the pLength specified. The data may be sent in a data flit, or an 8- or 32-byte data field specified for some control flits. 000 1 byte. Reserved when the command is an **amo***. 001 2 bytes. Reserved when the command is an **amo***. 010 4 bytes. Reserved when the command is **amo_rw** and the operation is specified as a Fetch and swap. That is the command flag is {x'8'..x'A'}. 011 8 bytes. Reserved when the command is **amo_rw** and the operation is specified as a Fetch and swap. That is the command flag is {x'8'..x'A'}. 100 16 bytes. Reserved when the command is an **amo***. 101 32 bytes. Reserved when the command is an **amo***. 110 Specifies 4-byte operands when the command is **amo_rw** and the operation is specified as a Fetch and swap. That is, the command flag is {x'8'..x'A'}. Otherwise, this field is reserved. 111 Specifies 8-byte operands when the command is **amo_rw** and the operation is specified as a Fetch and swap. That is, the command flag is {x'8'..x'A'}. Otherwise, this field is reserved.

Table 2-1. TL and TLX command operands (Page 5 of 5)

Operand mnemonic	Field width	Description
Resp_code	4	Response code. On a failed transaction, this field is found in a response packet reporting the reason the transaction failed. See the response packet for encoding and specifications. "Done" is not typically an included encoding because the response packet used is different for a failed transaction. For example, in response to a **rd_wnitc** AP command, the **read_failed** (TL response) is sent when the read is not able to complete successfully. The **read_response** (TL response) is used to indicate a successful completion and that data is associated with the response. A response code of "done" is implied with the **read_response**.
stream_id	4	Stream identifier used by the AP. This is used as part of the virtual channel, virtual queue, service queue specification.
T	1	Configuration read or write command type. 0 Indicates a type 0 configuration read or write command. A **config_write**, T=0 shall be used by the AFU to learn its bus number. For **config_read** with TL=0, the bus number is unchecked. 1 The operation shall result in a **mem_wr_fail** or **mem_rd_fail** TLX response with a Resp_code = Failed.

2.1 Handling multiple responses to a single command

As noted in the description of some responses, a single command may receive multiple responses. This might be due to a mismatch between the host's and OpenCAPI device's maximum data length specification.

For example, the host's or the device's internal bus protocol might be limited to atomically accessing 64 bytes of data. Read and write cases are examined in the following sections.

2.1.1 TLX Read request getting multiple TL responses

A 128-byte read request by the OpenCAPI device may be broken into two 64-byte read requests on the host protocol bus. This results in two TL responses returning data to the OpenCAPI device. The responses are not returned in any specified order. After all the responses are returned to the requester (the OpenCAPI device in this example), the requester examines the responses.

- When all responses indicate success, the command has completed successfully. In this example, each response provides 64 bytes of data, fulfilling the OpenCAPI device's request for 128 bytes.
- When all responses indicate failure, the command has failed. In this example, no data has been returned.
- When one response indicates success and the other indicates a failure, the command has failed. In this example, only 64 bytes of the requested 128 bytes have been returned. Because this is a read request, the data may be discarded. Depending on the Resp_code and the TL response, the entire operation, or just the failing portion may be retried. Refer to the specification of the TL response for when the operation may be retried.

The following TLX read commands may receive multiple responses.

- **rd_wnitc**, **rd_wnitc.n**

These read commands, when receiving multiple TL responses, shall see only the following responses[2]:

- **read_response**, **read_response.ow**, **read_failed**

[2] Not all responses apply to all commands. See the command descriptions for applicable responses.

2.1.2 TLX Write request getting multiple TL responses

A 128-byte write request by an OpenCAPI device may be broken into two 64-byte operations on the host protocol bus. This results in two TL responses indicating the status of the write operation in the host. The responses are not returned in any specified order. After all the responses are returned to the device, the device examines the responses.

- When all responses indicate success, the command has completed successfully. The write operation has completed and the changes to the specified memory locations are globally visible.
- When all responses indicate failure, the command has failed. The locations in memory specified by the command may have been modified by the failed operation. That is, the data at the locations may be unmodified, may contain undefined data, or may contain SUE data. The Resp_code field in the fail response indicates what might have occurred at the memory location specified by the write command.
- When one response indicates success and the other indicates failure, the command has failed. Only the data corresponding to the 64-byte block specified by the successful response has completed its operation in the host and the changes to the specified memory location are globally visible. The data corresponding to the 64-byte block specified by the failed response shall contain SUE data. Depending on the Resp_code and the TL response, the failing portion may be retried and the successful portion shall not be retried. Refer to the specification of the TL response for when the operation may be retried.

The following TLX write commands may receive multiple responses:

- **dma_w**, **dma_w.n**

These write commands, when receiving multiple TL responses, shall see only the following responses.

- **write_response**, **write_failed**

2.1.3 TL read request getting multiple TLX responses.

A 128-byte read request by the host to an OpenCAPI device may be broken into two 64-byte read requests at the OpenCAPI device. This results in two TLX responses returning data to the host. Further, the responses are not returned in any specified order. After all responses are returned to the host, the host examines the responses.

- When all responses indicate success, the command has completed successfully. In this example, each response provides 64-bytes of data, fulfilling the host’s request for 128 bytes.
- When all responses indicate failure, the command has failed. No data has been returned.
- When one response indicates success and the other indicates failure, the command has failed. In this example, only 64-bytes of the requested 128 bytes have been returned. The data obtained may be discarded. Depending on the Resp_code and the TLX response, the entire operation or just the failing portion may be retried.

The following TL read commands may receive multiple responses:

- **rd_mem**

These read commands, when receiving multiple TLX responses, shall see only the following responses:

- **mem_rd_response**, **mem_rd_response.ow**, **mem_rd_fail**

2.1.4 TL write request getting multiple TLX responses

A 128-byte write request by the host to an OpenCAPI device may be broken into two 64-byte write requests at the OpenCAPI device. This results in two TLX responses indicting the completion status of the write operation in the OpenCAPI device. Further, the responses are not returned in any specified order. After all the responses are returned to the host, the host examines the responses.

- When all responses indicate success, the command has completed successfully. The write operation has completed and the changes specified by the memory locations are globally visible,
- When all responses indicate failure, the command has failed. The locations in memory specified by the command may have been modified by the failed operation. That is, the data at the locations may be unmodified, may contain undefined data, or may contain SUE data. The Resp_code field in the fail response indicates what might have occurred at the memory location specified by the write command.
- When one response indicates success and the other indicates failure, the command has failed. Only the data corresponding to the 64-byte block specified by the successful response has completed its operation in the OpenCAPI device and the changes to the specified memory location are globally visible. The data corresponding to the 64-byte block specified by the failed response may be unmodified, may contain undefined data, or may contain SUE data. The contents of the data block is dependent on the address of the command. Depending on the Resp_code and the TLX response, the entire operation or just the failing portion may be retried. Refer to the specification of the TL response for when the operation may be retried and the state of the data block when the response indicates a failure.

The following TL write commands may receive multiple responses:

- **write_mem**

These commands, when receiving multiple TLX responses, shall see only the following responses:

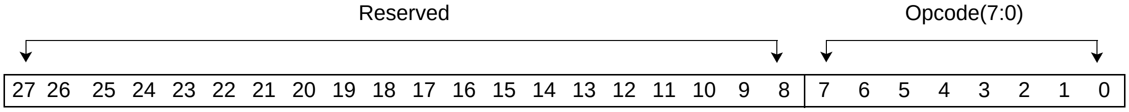
- **mem_wr_response**, **mem_wr_fail**.

2.2 TL CAPP command packets

TL commands are sent from the host to the AFU. An alphabetical list of the TL commands follows; each command is hyperlinked to its specification. In this section, the TL command specifications are in opcode order.

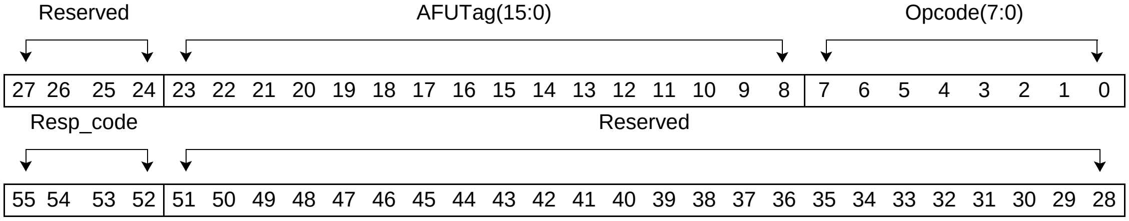
config_read	**config_write**	**intrp_rdy**	**mem_cntl**
nop	**pad_mem**	**pr_rd_mem**	**pr_wr_mem**
rd_mem	**rd_pf**	**write_mem**	**write_mem.be**
xlate_done			

No operation	**nop**	‘0000 0000’
NA	NA	1



This command has no operands and performs no action. It is discarded at the TLX.

Address translation completed	**xlate_done**	‘0001 1000’
async notification	TL.vc.0	2



The host is sending an asynchronous notification that an address translation requested by a prior TLX command has completed with the indicated response code. The remaining fields of the command identify the prior TLX command.

The TL is required to maintain the order of the matching **read_failed**, **write_failed,** or **touch_resp** response packets carrying the AFUTag and the Resp_code = intrp_pending when loading the VC. That is, the TL shall ensure that the response packets precede the **xlate_done** command in the VC.

The following illustrates how **xlate_done** is used:

1. The device issues a command that requires an address translation that the host is unable to complete.
2. The host responds with
   a. a **read_failed** response with a Resp_code = xlate_pending. See *Table 2-11* ***read_failed resp_code use by tlx command*** on page 76 for a list of the commands the device might have issued in step 1.

   b. a **write_failed** response with a Resp_code = xlate_pending. See *Table 2-13* ***write_failed resp_code use by tlx command*** on page 79 for a list of the commands the device might have issued in step 1.

   c. a **touch_resp** response with a Resp_code = xlate_pending. See *Table 2-9* ***touch_resp resp_code use by tlx command*** on page 73 for a list of the commands the device might have issued in step 1.

3. Once the host has completed the address translation, the host issues **xlate_done** indicating if the device should retry or abort the operation.

The Resp_code field is specified in *Table 2-2*.

Table 2-2. The Resp_code specification for ***xlate_done***

Resp_code encode	Description
‘0000’	Completed. Address translation completed successfully
‘0001’	Reserved.
‘0010’	Retry request (rty_req). Indicates that the address translation could not be completed at this time. The AFU may make an address translation attempt at a later time. This is a long back-off event.
‘0011’ - ‘1110’	Reserved.
‘1111’	Translation address error (adr_error). Indicates that the address translation requested resulted in an address translation error.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

This command is posted.

Interrupt ready	**intrp_rdy**	‘0001 1010’
async notification	TL.vc.0	2

Reserved	AFUTag(15:0)	Opcode(7:0)
27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

Resp_code	Reserved
55 54 53 52	51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28

The host is sending an asynchronous status notification for a previously attempted interrupt. The AFU determines its actions based on the Resp_code received. The AFUTag field of the command identifies the prior TLX command.

The TL is required to maintain the order of the matching **intrp_resp** carrying the AFUTag and the Resp_code = intrp_pending when loading the VC. That is, the TL shall ensure that the response packets precede the **intrp_rdy** command in the VC.

This command is used by

1. **intrp_req**:

   a. The device issues an **intrp_req** using the *cmd_flag* and *Object_handle* specified in the device’s MMIO space.

b. The host protocol, using the *Object_handle* for some form of address translation, is unable to complete. The host returns an **intrp_resp** with a Resp_code of intrp_pending (4).

c. Once the host has completed the address translation, the host issues **intrp_rdy** indicating if the device should retry or abort the **intrp_req** operation.

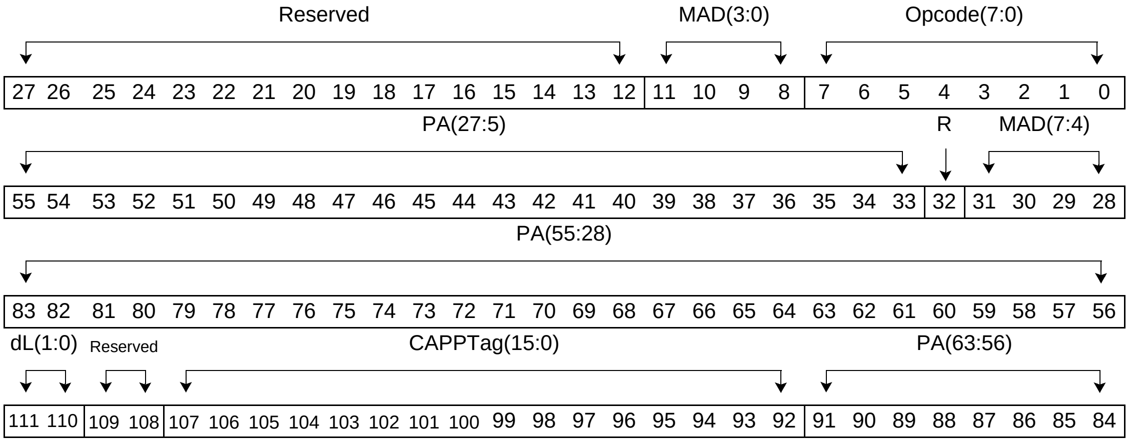
The Resp_code field is specified in *Table 2-3*.

Table 2-3. The Resp_code specification for ***intrp_rdy***

Resp_code encode	Description
‘0000’	Ready to service the interrupt. The AFU may retry the prior **intrp_req**, or **intrp_req.d** command.
‘0001’	Reserved.
‘0010’	Retry request (rty_req). Indicates that the host is unable to service the interrupt at this time. The AFU may retry the prior interrupt request at a later time as specified by its long back off event timer. This is a long back-off event.
‘0011’ - ‘1101’	Reserved.
‘1110’	Failed. The host is unable to service the interrupt specified by the prior command. Any future attempt specifying the same interrupt parameters shall fail. **Engineering Note** Note that **intrp_req** is specified in a way that the *cmd_flag* and *Object_handle* are host specific and the values used are found in the device’s configuration space. A device correctly using the *cmd_flag* and *Object_handle* should not normally see this Resp_code. A malicious device using values other than those provided in its MMIO space may see a failed Resp_code. It is strongly recommended that an implementation provide error collection facilities to indicate the reason for this Resp_code. The specification of the error collection facility should be documented in the host’s platform architecture.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

This command is posted.

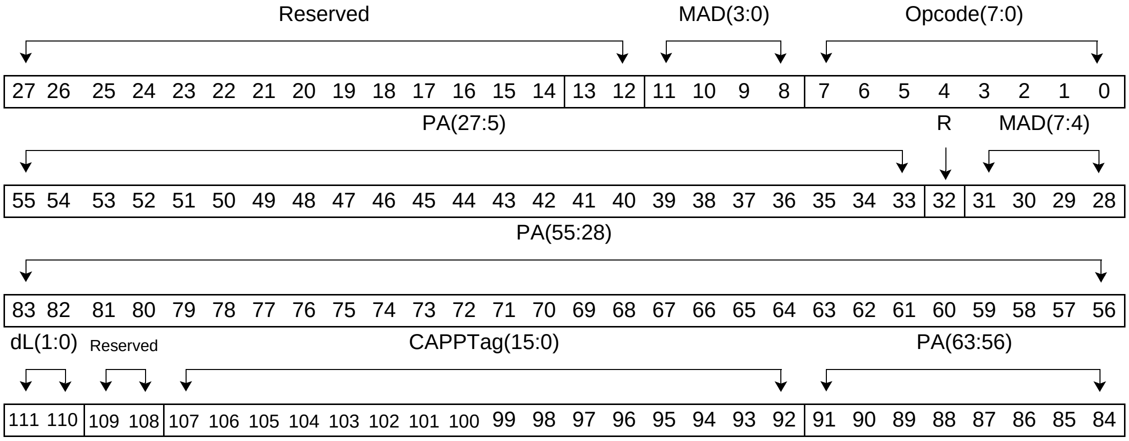
Read memory	**rd_mem**	‘0010 0000’
mem_read	TL.vc.1	4



The host is requesting data to be read from the AFU memory. The starting address specified by the PA, supports a *critical OW request*. The data is a *naturally aligned data block* with a length specified by the dLength field (dL). See the host’s platform architecture for the specification of the *MAD* field.

The response to this command is **mem_rd_response**, **mem_rd_response.ow**, or **mem_rd_fail**. The **mem_rd_fail** response indicates the operation failed. See the response packet for encoding and specifications.

Read Prefetch	**rd_pf**	‘0010 0010’
mem_read	TL.vc.1	4



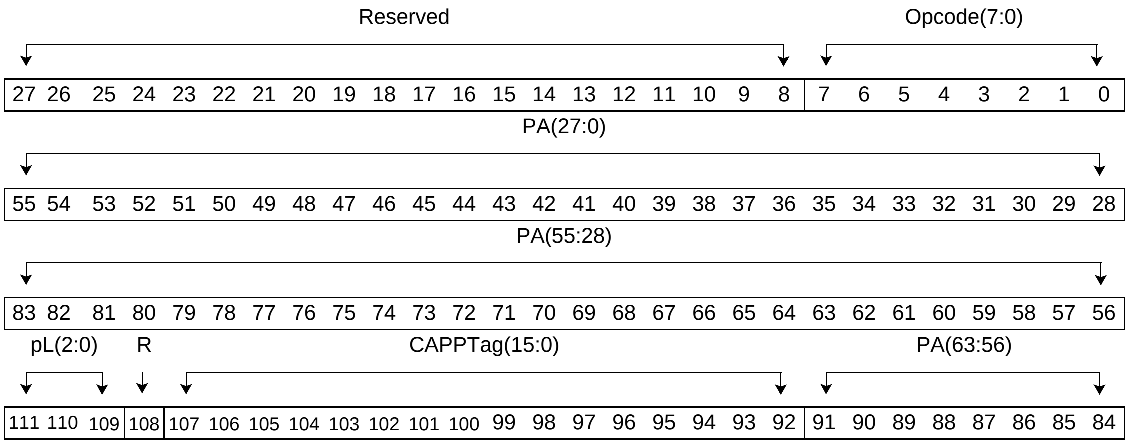
The host is requesting data to be prefetched by the AFU memory. The starting address, specified by the PA, supports the *critical OW request* format. The data requested is a *naturally aligned data block* with a length specified by the dLength field (dL). See the host’s platform architecture for the specification of the *MAD* field. No data is returned. The AFU$_M$, when provisioned, may hold data in temporary buffers. The command is intended to provide hints to the AFU$_M$ to access data with long access times that might be required at a later time. The host may or may not request the data at a later time.

The AFU_M may ignore the command’s prefetch hints. The architecture does not specify any errors for this command. Any event occurring in the AFU_M that prohibits the operation from completing is not reported and the command execution is aborted.

The command is posted.

> **Developer Note**
>
> If error checking were architected, the architecture would specify the same errors defined for **rd_mem**. Since there is no architected mechanism for a posted command to retry an operation, those types of Resp_code events specified by a response to a **rd_mem** are eliminated. Since data is not returned to the host, dError events are eliminated. This leaves only a Failed error.
>
> Prefetch mechanisms tend to be somewhat inaccurate. Data that is not required by the application might be requested for prefetch and is never demand fetched by the application. The architecture for **rd_pf** pushes any error detection onto a subsequent demand fetch (**rd_mem**) using the same PA.

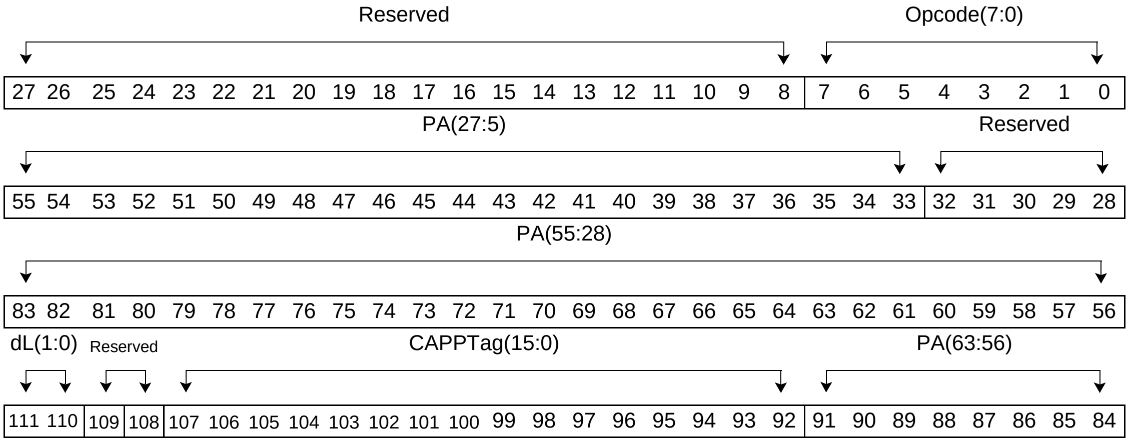
Partial memory read	**pr_rd_mem**	‘0010 1000’
pr_mem_read	TL.vc.1	4



The host is requesting data to be read from the AFU memory. The number of bytes transfered is specified by pLength field(pL), and the starting address shall be naturally aligned based on the number of bytes requested. The pLength field limits the transfer size to 2^n bytes where n = {0..5}.

The response to this command is **mem_rd_response**, **mem_rd_response.ow**, **mem_rd_response.xw**, or **mem_rd_fail**. When a **mem_rd_response** is received, the data is found in the 64-byte data flit (only one is returned for this command). The data is address aligned as specified in *Section 5.1.3 Data transport, order, and alignment* on page 105. When a **mem_rd_response.ow** is received, the data is found in the 32-byte data field specified by some control flits. The data is address aligned within the data field. The **mem_rd_response.xw** response may be used only when the data length specified by pL is 8 or fewer bytes. That is, when **mem_rd_response.xw** is used to return data, the data length specified by pL shall be 8 or fewer bytes. The **mem_rd_fail** response indicates the operation failed. See the response packet for encoding and specifications.

Pad memory	**pad_mem**	‘1000 0000’
mem_write	TL.vc.1	4



The host is issuing a request to load memory using a data pattern set during the configuration of the device. The configuration of the data pattern in the device is device implementation dependent and is beyond the scope of this architecture. The pattern is used to fill a 32-, 64-, 128- or 256-byte naturally aligned block of data in the AFU memory. The starting address is specified by the PA field and shall be naturally aligned based on the length of the data as specified by the dLength (dL) field[3].

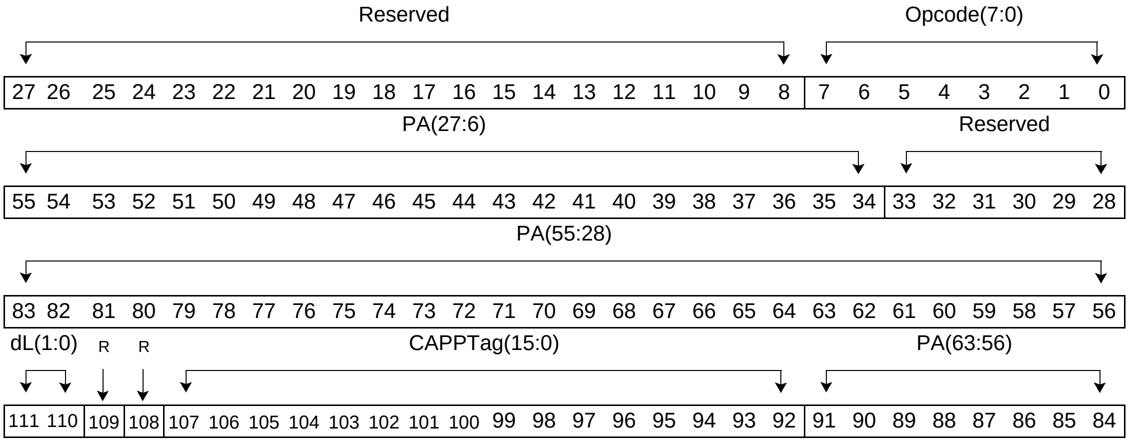
Engineering Note

The pattern is expected to be found through the device’s configuration space and may be located in the device’s MMIO space. The width of the pattern shall be a power of 2 number of bits. The pattern is applied starting at offset 0 (bit 0, byte 0) of the data block. When the pattern is smaller than the operation width specified by the dLength field, the pattern is repeated. When the pattern is larger than the operation width specified by the dLength field, the pattern starting at offset 0 is applied and as space permits offset 1 is applied and so on.

The **mem_wr_response** and **mem_wr_fail** responses to this command indicate the status of the pad memory operation. The **mem_wr_response** indicates a successful completion of the operation. The **mem_wr_fail** response indicates that the operation failed. See the response packet description for encoding and specifications.

[3] A *dLength* value of ‘00’ specifies a 32-byte data block.

Write memory	**write_mem**	‘1000 0001’
mem_write	TL.vc.1, TL.dcp.1	4

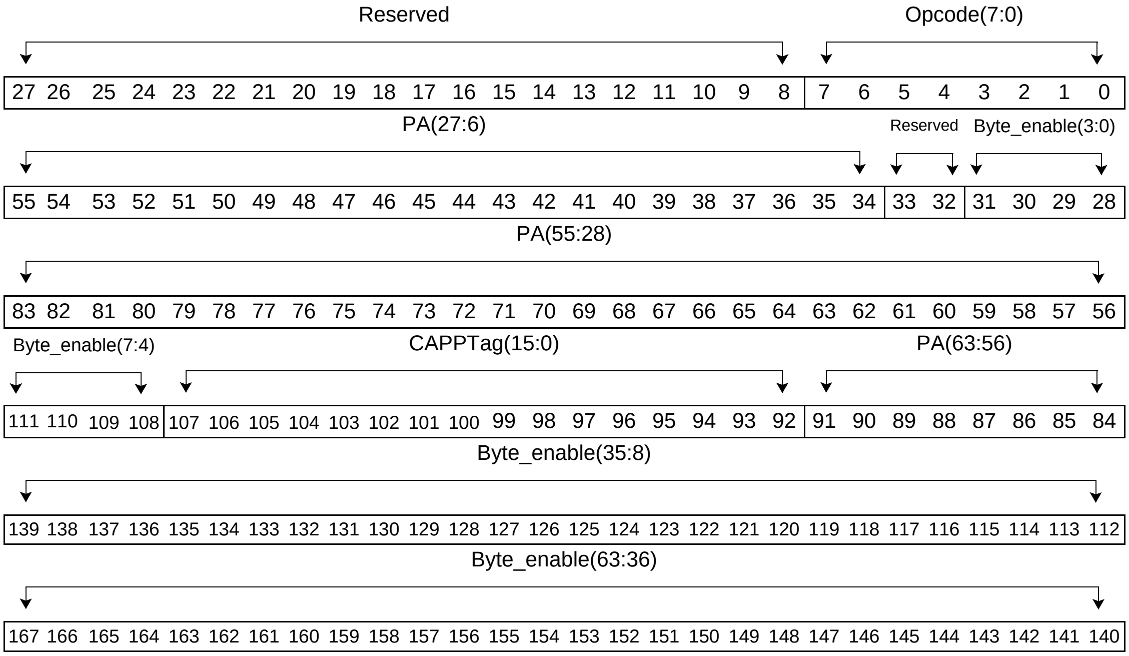


The host is writing a 64-, 128-, or 256-byte block of data to the AFU memory. The starting address is specified by the PA field and shall be naturally aligned based on the length of the data as specified by the dLength (dL) field.

This command is specified with immediate data. The data may be transfered using data flits or 32-byte data fields found within a control flit. Credits for both the VC and DCP shall be obtained before this command is serviced by the TL.

The **mem_wr_response** and **mem_wr_fail** responses to this command indicate the status of the write to memory operation. The **mem_wr_response** indicates a successful completion of the operation. The **mem_wr_fail** response indicates that the operation failed. See the response packet description for encoding and specifications.

Byte enable memory write	**write_mem.be**	‘1000 0010’
pr_mem_write	TL.vc.1, TL.dcp.1	6

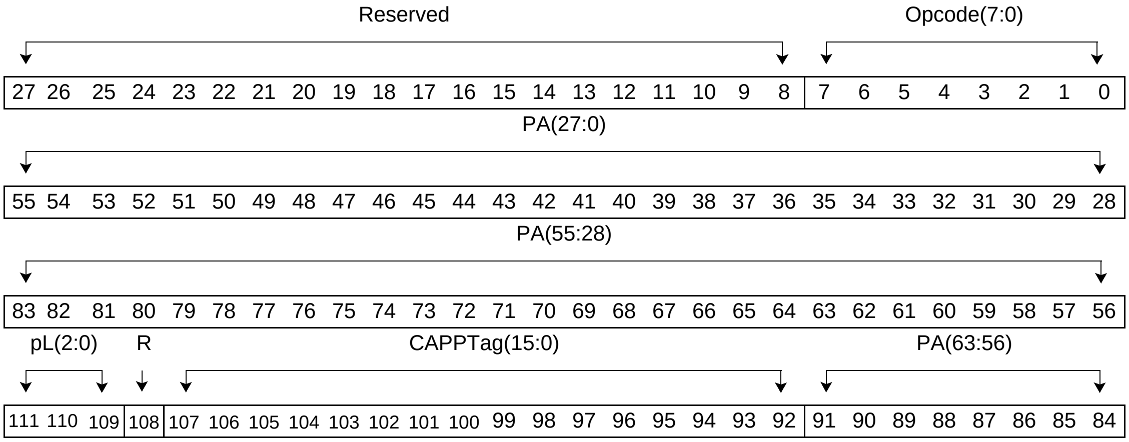


The host is writing a 64-byte data block to the AFU memory using a 64-bit byte-enable field. Each bit corresponds to one byte of the 64-byte aligned data block specified by the PA, where bit 0 of the BE determines if byte 0 of the data is written. When BE(n) is set to 1, byte n is written where n={0..63}.

This command is specified with immediate data. The data shall be sent using a single 64-byte data flit. Credits for both the VC and DCP shall be obtained before this command is serviced by the TL.

The **mem_wr_response** and **mem_wr_fail** responses to this command indicate the status of the write to memory operation. The **mem_wr_response** indicates a successful completion of the operation. The **mem_wr_fail** response indicates that the operation failed. See the response packet for encoding and specifications.

Partial cache line memory write	**pr_wr_mem**	‘1000 0110’
pr_mem_write	TL.vc.1, TL.dcp.1	4

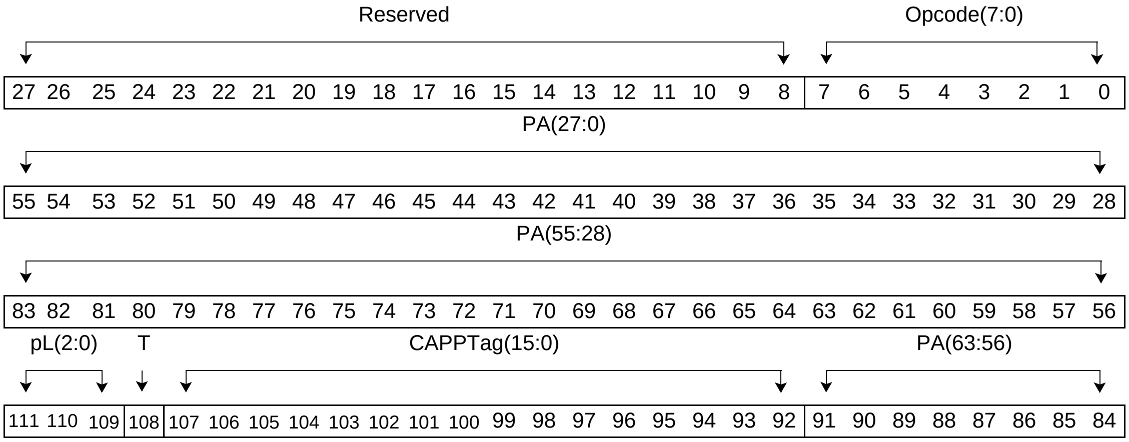


The host is writing the data to the AFU memory. The starting address is specified by the PA and shall be naturally aligned based on the length of the data as specified by the pLength (pL) field. The combination of the address and the pLength shall not cross a 64-byte address boundary.

This command is specified with immediate data. The data may be sent in a data flit, or it may be sent in a 32-byte data carrier. When the pL field indicates a length of 8 bytes or less, the data may be sent in an 8-byte data carrier. The data is address aligned as specified in *Section 5.1.3 Data transport, order, and alignment* on page 105. Credits for both the VC and DCP shall be obtained before this command is serviced by the TL.

The **mem_wr_response** and **mem_wr_fail** responses to this command indicate the status of the write to memory operation. The **mem_wr_response** indicates a successful completion of the operation. The **mem_wr_fail** response indicates that the operation failed. See the response packet for encoding and specifications.

Configuration read	**config_read**	‘1110 0000’
configuration	TL.vc.1	4



The host is issuing a read to the AFU’s configuration address space. The number of bytes transfered is specified by pLength (pL) field. The starting address shall be naturally aligned based on the number of bytes requested. For this command, the pLength value is limited to a transfer size of 1, 2, or 4 bytes. That is, the specification of pLength shall limit the transfer size to 2^n bytes where n = {0..2}.

The PA field is defined as follows:

PA bits	Description
63:32	Reserved. Shall be set to $^{32}0$.
31:24	Bus number (7:0).
23:19	Device number (4:0).
18:16	Function number (2:0).
15:12	Reserved. Shall be set to $^{4}0$.
11:2	Register number.
1:0	Byte offset within the register.

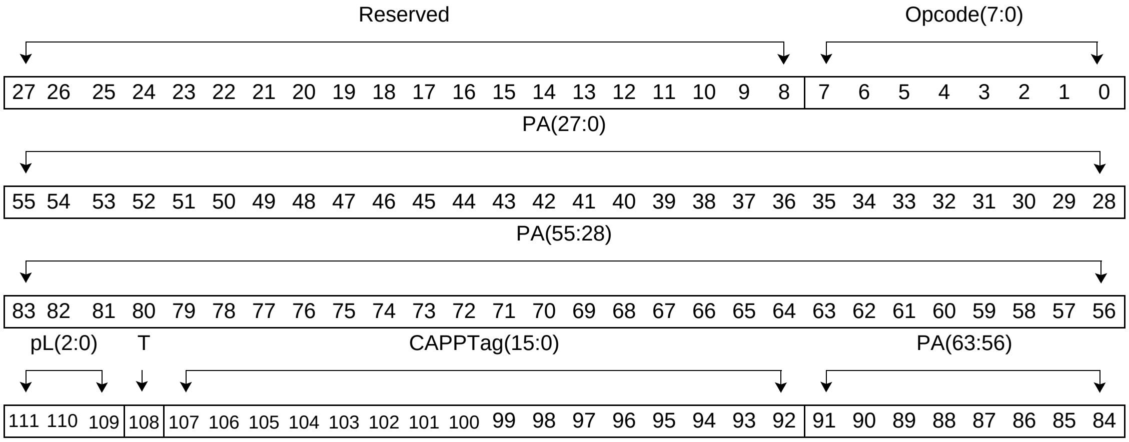
The T field, defined in *Table 2-1* on page 31, specifies the configuration type of the command.

The response to this command is **mem_rd_response**, **mem_rd_response.ow**, **mem_rd_response.xw**, or **mem_rd_fail**. When a **mem_rd_response** is received, the data is found in the 64-byte data flit (only one is returned for this command). The data is address aligned as specified in *Section 5.1.3 Data transport, order, and alignment* on page 105. When a **mem_rd_response.ow** is received, the data is found in the 32-byte data field found in a control flit (only one is returned for this command). The data is address aligned. When a **mem_rd_response.xw** is received, the data is found in the 8-byte data field found in a control flit and is address aligned.

Neither metadata or extended-metadata are specified for this command. If the template used when returning the response specifies metadata or extended-metadata for the data carrier used, the metadata and extended-metadata is discarded by the host, an error shall not be reported, and the operation completes successfully.

The **mem_rd_fail** response indicates the operation failed. If the device or function number are not recognized by the AFU, the operation shall fail with a Resp_code = Failed. See the response packet for encoding and specifications.

Configuration write	**config_write**	‘1110 0001’
configuration	TL.vc.1, TL.dcp.1	4



The host is issuing a write to the AFU’s configuration address space. The number of bytes transfered is specified by pLength (pL) field. The starting address shall be naturally aligned based on the number of bytes requested. For this command, pLength is limited to a transfer size of 1, 2, or 4 bytes. That is, the specification of pLength shall limit the transfer size to 2^n bytes where n = {0..2}.

The PA field is defined as follows:

PA bits	Description
63:32	Reserved. Shall be set to ^{32}0.
31:24	Bus number (7:0).
23:19	Device number (4:0).
18:16	Function number (2:0).
15:12	Reserved. Shall be set to ^{4}0.
11:2	Register number.
1:0	Byte offset within the register.

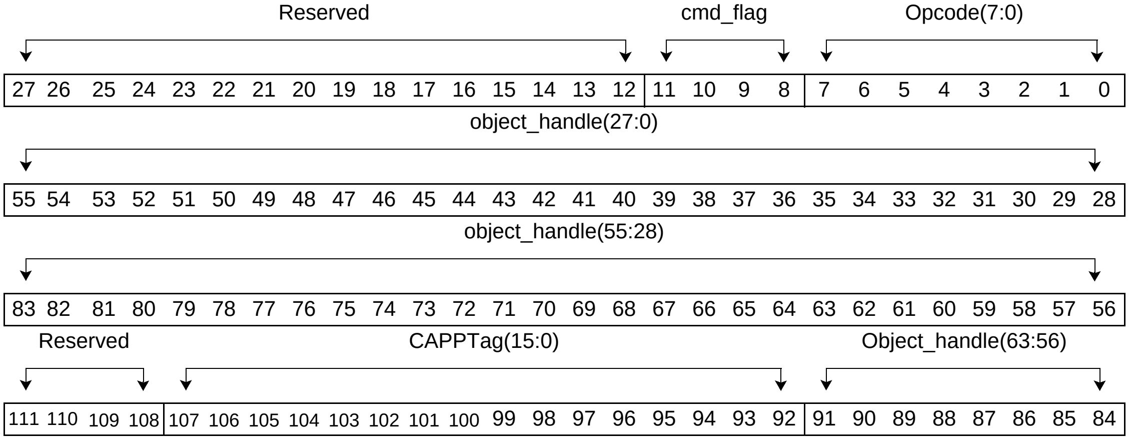
The T field, defined in *Table 2-1* on page 31, specifies the configuration type of the command. When T is set to 0, the AFU learns its bus number located in the PA field.The device and function number are assigned by the attached OpenCAPI device and are not modified by any configuration actions. If the device or function numbers are not recognized, the operation shall fail and the data is discarded. The failure shall be reported using a TLX **mem_wr_fail** response with a Resp_code= Failed.

This command is specified with immediate data. The data is address aligned as specified in *Section 5.1.3 Data transport, order, and alignment* on page 105. The data may be sent in a data flit, or it may be sent in a 32- or 8-byte data carrier. Credits for both the VC and DCP shall be obtained before this command is serviced by the TL.

Neither metadata or extended-metadata are specified for this command. If the template used when issuing this command specifies metadata or extended-metadata for the data carrier used, the metadata shall be discarded by the AFU, an error shall not be reported, and the operation completes successfully.

The **mem_wr_response** and **mem_wr_fail** responses to this command indicate the status of the write to memory operation. The **mem_wr_response** indicates a successful completion of the operation. The **mem_wr_fail** response indicates that the operation failed. See the response packet for encoding and specifications.

Memory control	**mem_cntl**	‘1110 1111’
message	TL.vc.0	4



This command is used for device defined functions specified by the device manufacturer. The cmd_flag and object_handle are defined by the device manufacturer. Any errors in the host’s specification of these fields results in the operation failing.

The response to this command is a **mem_cntl_done**.

2.3 TLX AP command packets

TLX commands are sent from the AFU to the host. An alphabetical list of the TLX commands follows; each command is hyperlinked to its specification. In this section, the TLX command specifications are in opcode order.

amo_rd	**amo_rd.n**	**amo_rw**	**amo_rw.n**
amo_w	**amo_w.n**	**assign_actag**	**dma_pr_w**
dma_pr_w.n	**dma_w**	**dma_w.be**	**dma_w.n**
dma_w.be.n	**intrp_req**	**intrp_req.d**	**nop**
pr_rd_wnitc	**pr_rd_wnitc.n**	**rd_wnitc**	**rd_wnitc.n**
wake_host_thread	**xlate_touch**	**xlate_touch.n**	

No operation	**nop**	‘0000 0000’
NA	NA	1

Reserved	Opcode(7:0)
27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

This command has no operands and performs no action. It is discarded at the TL.

Read with no intent to cache	**rd_wnitc** **rd_wnitc.n**	‘0001 0000’ ‘0001 0100’
dma_read	TLX.vc.3	4

stream_id(3:0)	acTag(11:0)	MAD(3:0)	Opcode(7:0)
27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0

EA(27:5)	R	MAD(7:4)
55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33	32	31 30 29 28

EA(55:28)
83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56

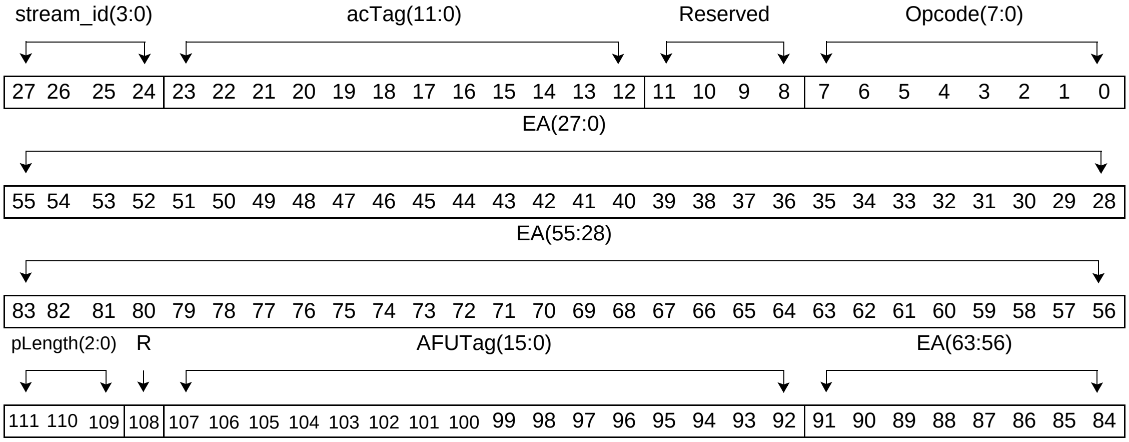
dL(1:0)	Reserved	AFUTag(15:0)	EA(63:56)
111 110	109 108	107 106 105 104 103 102 101 100 99 98 97 96 95 94 93 92	91 90 89 88 87 86 85 84

The AFU is requesting to read data with no intent to cache. The starting address specified by the EA supports a *critical OW request*. The data is a *naturally aligned data block* with a length specified by the dLength field (dL). See the host’s platform architecture for the specification of the *MAD* field.

- The dot-n form indicates that the results of the address translation may not be installed into the host's ATC as part of ATC miss handling.

The response to this command is **read_response**, **read_response.ow**, or **read_failed**. Multiple responses to a single **rd_wnitc** may occur. The **read_failed** response indicates the operation failed. See the response packet for additional details.

Partial read with no intent to cache	**pr_rd_wnitc** **pr_rd_wnitc.n**	'0001 0010' '0001 0110'
pr_dma_read	TLX.vc.3	4

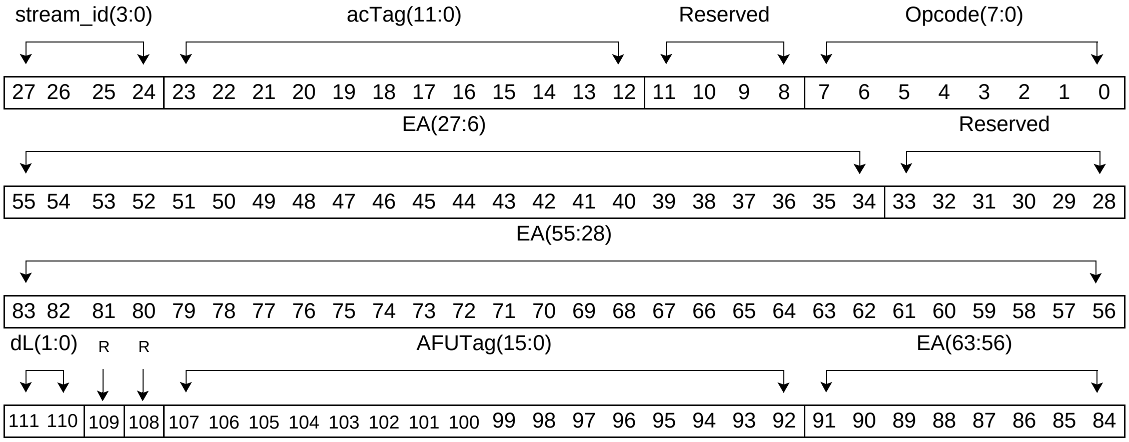


The AFU is requesting to read a partial cache line of data with no intent to cache at the address specified by the EA. The starting address shall be naturally aligned based on the length of the data specified by the pLength field. The pLength restricts this command to lengths of powers of 2 ranging from 1 to 32 bytes.

- The dot-n form indicates that the results of the address translation may not be installed into the host's ATC as part of ATC miss handling.

The response to this command is **read_response**, **read_response.ow**, or **read_failed**. When a **read_response** is received, the data is address aligned (address bits 5:0) in the 64-byte data flit (only one is returned for this command). When a **read_response.ow** is received, the data is address aligned (address bits(4:0)) in the 32-byte data carrier (only one is returned for this command). The **read_failed** response indicates the operation failed. See the response packet for additional details.

DMA Write	**dma_w** **dma_w.n**	‘0010 0000’ ‘0010 0100’
dma_write	TLX.vc.3, TLX.dcp.3	4



The AFU is requesting to write data at the address specified by the EA.The starting address is specified by the EA and shall be naturally aligned based on the length of the data as specified by the dLength field.

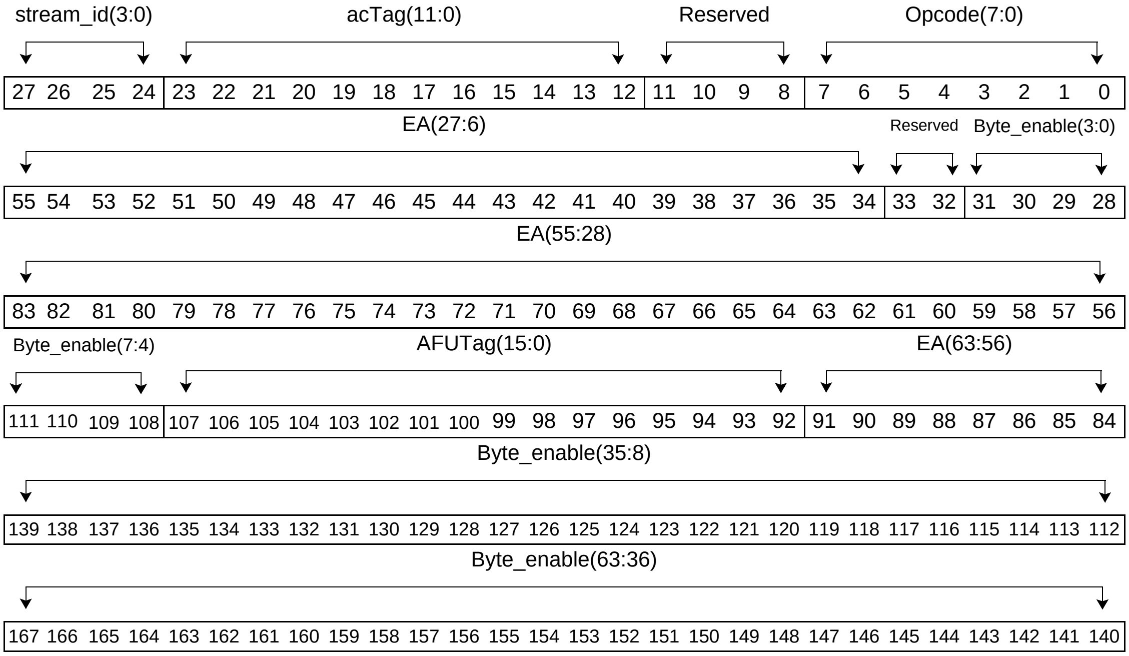
This command is specified with immediate data. The data may be sent in a data flit, or it may be sent in multiple 32-byte data carriers. Credits for both the VC and DCP shall be obtained before this command is serviced by the TLX.

The AFU TLX shall not service this command unless all data specified by dLength is available to be sent.

- The dot-n form indicates that the results of the address translation may not be installed into the host’s ATC as part of ATC miss handling.

The host shall respond with either a **write_response** or a **write_failed** response packet. The **write_failed** response indicates that the operation failed. See the response packet description for additional details.

Byte enable DMA Write	**dma_w.be** **dma_w.be.n**	‘0010 1000’ ‘0010 1100’
pr_dma_write	TLX.vc.3, TLX.dcp.3	6



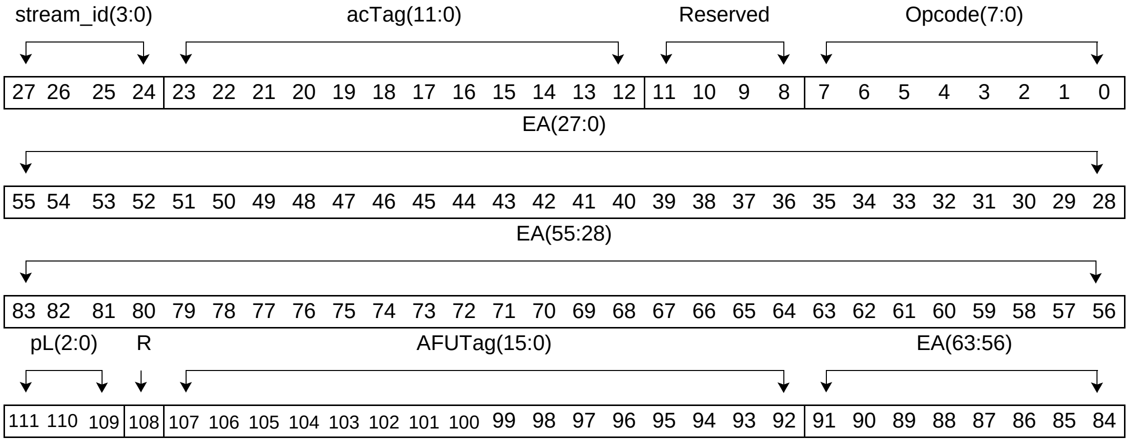
The AFU is writing data at the address specified by the EA using a 64-bit byte-enable field. Each bit corresponds to one byte of the 64-byte aligned data block specified by the EA, where bit 0 of the BE determines if byte 0 of the data is written. When BE(n) is set to 1, byte n is written where n={0..63}.

This command is specified with immediate data. The data shall be sent in a 64-byte data flit. Credits for both the VC and DCP shall be obtained before this command is serviced by the TLX.

- The dot-n form indicates that the results of the address translation may not be installed into the host’s ATC as part of ATC miss handling.

The host shall respond with either a **write_response** or a **write_failed** response packet. The **write_failed** response indicates that the operation failed. See the response packet description for additional details.

DMA parital write	**dma_pr_w** **dma_pr_w.n**	‘0011 0000’ ‘0011 0100’
pr_dma_write	TLX.vc.3, TLX.dcp.3	4



The AFU is requesting to write data starting at the address specified by the EA. The starting address shall be naturally aligned based on the length of the data as specified by the pLength (pL) field. The pLength restricts this command to lengths of powers of 2 ranging from 1 to 32 bytes. The combination of the EA and the pLength shall not cross a 64-byte address boundary.

Only a single data carrier is associated with this command.

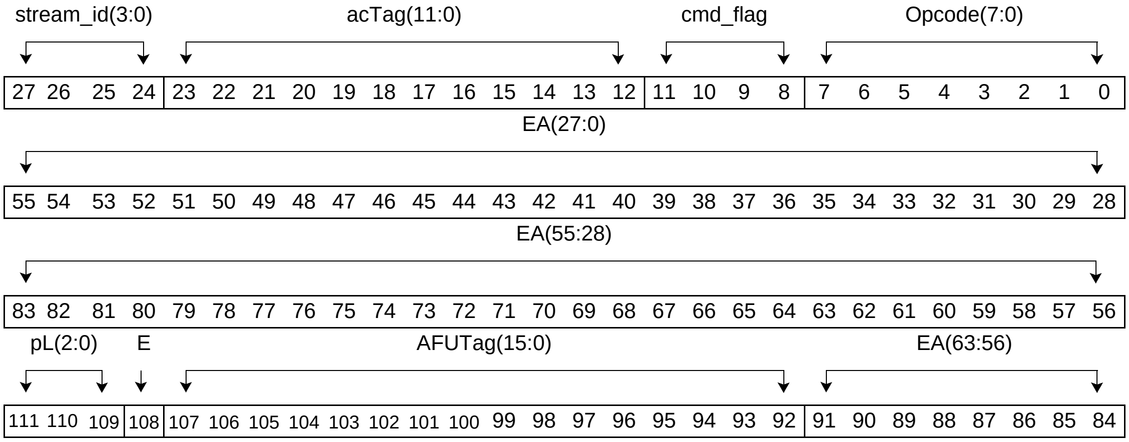
This command is specified with immediate data. The data is address aligned as specified in *Section 5.1.3 Data transport, order, and alignment* on page 105. The data may be sent in a data flit, or it may be sent in a 32-byte data carrier. When the length specified by pL is 8 or less bytes, the data may be sent in an 8-byte data carrier. Credits for both the VC and DCP shall be obtained before this command is serviced by the TLX.

The AFU TLX shall not service this command unless all data specified by pLength is available to be sent.

- The dot-n form indicates that the results of the address translation may not be installed into the host’s ATC as part of ATC miss handling.

The host shall respond with either a **write_response** or a **write_failed** response packet. The **write_failed** response indicates that the operation failed. See the response packet description for additional details.

AMO read	**amo_rd** **amo_rd.n**	‘0011 1000’ ‘0011 1100’
atomics.r	TLX.vc.3	4



The AFU is requesting an atomic memory operation specified by the cmd_flag. All operands for this request are found in memory as specified by the EA.

- The dot-n form indicates that the results of the address translation may not be installed into the host’s ATC as part of ATC miss handling.

There shall be a single response to this command. The response to this command shall be one of the following TL responses: **read_response**, **read_response.ow**, **read_response.xw**, or **read_failed**. The **read_failed** response indicates that the operation failed. See the response packet specification for additional details.

Operation:

The operand length, as specified by the pLength (pL) field is restricted to 4- and 8-byte operands. That is, the pLength shall be specified as {‘010’, ‘011’}; all other values are reserved. Two signed integer operands are specified. The first operand “A” is found at the address specified by the command. The second operand “A2” is found at the address specified with an offset specified by the width of the operands and the operation; that is, as specified by pLength and by the command flag.

- For Fetch and increment bounded and Fetch and increment equal (that is, cmd_flag = {‘1100’, ‘1101’)), A2 is found at the address specified *plus* the width of the operand.
- For Fetch and decrement bounded (that is, cmd_flag = {‘1110’}), A2 is found at the address specified *minus* the width of the operand.

The specification of the address is constrained to be naturally aligned. In addition:

- It cannot target locations at 32n-$2^{\text{bin2dec(pL)}}$, where n= 1, 2, 3... (Fetch and increment bounded and Fetch and increment equal; that is, cmd_flag = {‘1100’, ‘1101’}).
- It cannot target locations at 32n, where n = 0, 1, 2, 3... (Fetch and decrement bounded; that is, cmd_flag = {‘1110’}).

The original value from the memory location specified by the command, or the 4- or 8-byte minimum signed integer value, is returned with **read_response**.

The operation performed is specified by the cmd_flag. The endianness of the operands is specified by the E bit.

*Table 2-4. The cmd_flag specification for **amo_rd***

cmd_flag	Operation name and description
‘0000’ through ‘1010’	Reserved

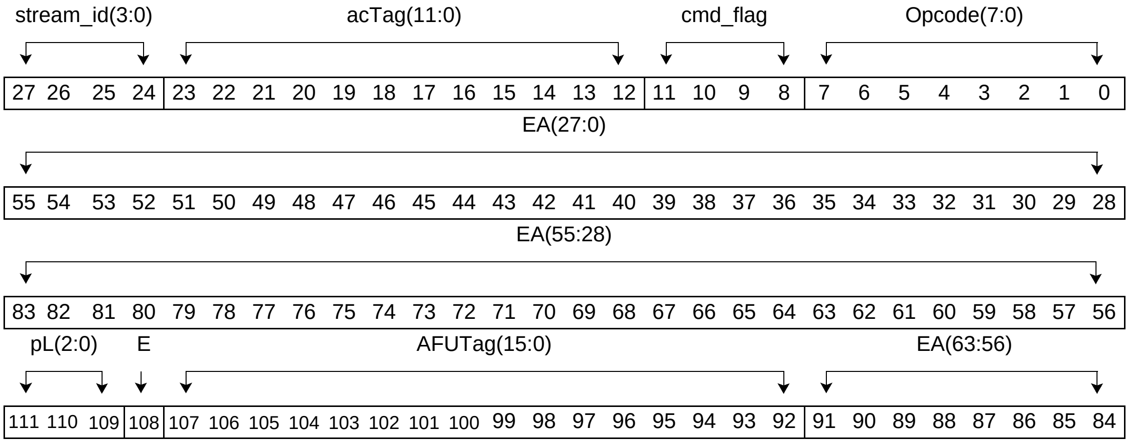
‘1100’	**Fetch and increment bounded**
	t ← A;
	If A != A2 then {A ← A+1; return t}
	else {return minimum signed integer value}

‘1101’	**Fetch and increment equal**
	t ← A;
	If A = A2 then {A ← A+1; return t}
	else {return minimum signed integer value}

‘1110’	**Fetch and decrement bounded**
	t ← A;
	If A != A2 then {A ← A-1; return t}
	else {return minimum signed integer value}

‘1111’	Reserved

AMO read write	**amo_rw** **amo_rw.n**	‘0100 0000’ ‘0100 0100’
atomics.rw	TLX.vc.3, TLX.dcp.3	4



The AFU is requesting an atomic memory operation specified by the cmd_flag. For this request, operands are provided with the command and are found in memory as specified by the EA.

- The dot-n form indicates that the results of the address translation may not be installed into the host’s ATC as part of ATC miss handling.

This command is specified with immediate data. The data may be sent using data flits, 32-byte data carriers, or 8-byte data carriers as described in the following operation description. Use of 8-byte data carriers is restricted as specified below. Credits for both the VC and DCP shall be obtained before this command is serviced by the TLX.

The AFU TLX shall not service this command unless all data specified by pLength is available to be sent.

There shall be a single response to this command. The response to this command shall be one of the following TL responses: **read_response**, **read_response.ow**, **read_response.xw**, or **read_failed**. The **read_failed** response indicates that the operation failed. See the response packet specification for additional details.

Operation:

The command address specified shall be naturally aligned based on the operand length. The operand length is restricted to 4- and 8-byte operands. The pLength (pL) shall be specified as {‘010’, ‘011’} for all cmd_flag operations with the exception of fetch and swap operations where the cmd_flag is specified as {x‘8’...x‘A’} and pLength shall be specified as {‘110, ‘111’}. Refer to the specification of *pLength on page 34*.

Operations specified by the cmd_flag use either two or three operands; additional classification of the operands can be found in the description of the operation in *Table 2-5*. The command’s address specifies the location of a first operand, “A”.

Operand A is operated on by the second operand, “V”, which is provided as the command’s write data. Operand V is aligned within one of the following:

- a 64-byte data flit based on address bits 5:0 specified by the command.
- a 32-byte data field carried in a control flit based on address bits 4:0 specified by the command.
- an 8-byte data field carried in a control flit based on address 2:0 specified by the command. This option shall not be used for fetch and swap operations where the cmd_flag is specified as {x‘8’...x‘A’}.

A third operand “W” is provided for fetch and swap operations. Operand W is placed in the same data carrier as operand V. Operand W shall be aligned within one of the following:

- the 64-byte data flit based on the following equation:

  alignment (5:0) ← EA(5:4) || (EA(3:0) + ‘1000’)
  Any carryout from bit 3 is ignored.

- the 32-byte data field carried in a control flit based on the following equation:

  alignment (4:0) ← EA(4) || (EA(3:0) + ‘1000’)
  Any carryout from bit 3 is ignored.

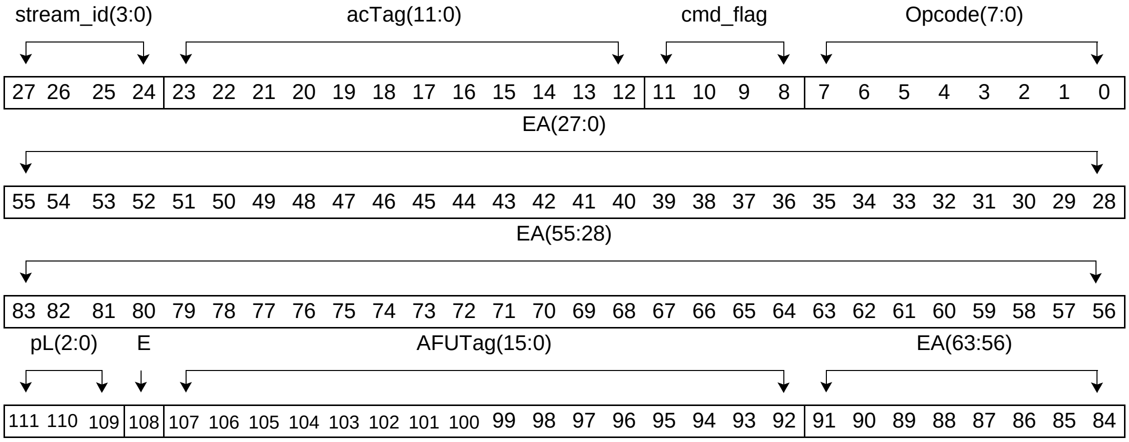
The original value from the memory location specified by the command shall be returned with a **read_response**, **read_response.ow**, or **read_response.xw**.

The endianness of the operands is specified by the E bit. The value of E might not affect the result of the operation specified by the cmd_flag. This is noted in the operation description found in *Table 2-5*.

Table 2-5. The cmd_flag specification for ***amo_rw***

cmd_flag	Operation name and description
‘0000’	**Fetch and Add** Operands are unsigned integers. t ← A; A ← A + V; return t Overflow conditions are not reported.
‘0001’	**Fetch and XOR** Operands are bit vectors. E has no effect on the operation. t ← A; $A \leftarrow V \oplus A$; return t
‘0010’	**Fetch and OR** Operands are bit vectors. E has no effect on the operation. t ← A; $A \leftarrow V \vee A$; return t
‘0011’	**Fetch and AND** Operands are bit vectors. E has no effect on the operation. t ← A; $A \leftarrow V \wedge A$; return t
‘0100’	**Fetch and maximum unsigned** Operands are unsigned integers. t ← A; A ← Max(A, V); return t A is unchanged when A is greater than or equal to V.
‘0101’	**Fetch and maximum signed** Operands are signed two’s complement integers. t ← A; A ← Max(A, V); return t A is unchanged when A is greater than or equal to V.
‘0110’	**Fetch and minimum unsigned** Operands are unsigned integers. t ← A; A ← Min(A, V); return t A is unchanged when A is less than or equal to V.
‘0111’	**Fetch and minimum signed** Operands are signed two’s complement integers. t ← A; A ← Min(A, V); return t A is unchanged when A is less than or equal to V.
‘1000’	**Fetch and swap** Operands are bit vectors. E has no effect on the operation. V is not used. t ← A; A ← W; return t
‘1001’	**Fetch and swap equal** Operands are bit vectors. E has no effect on the operation. t ← A; When V = A, then A ← W; return t
‘1010’	**Fetch and swap not equal** Operands are bit vectors. E has no effect on the operation. t ← A; when $V \neq A$, then A ← W; return t
‘1011’ through ‘1111’	Reserved

AMO write	**amo_w** **amo_w.n**	‘0100 1000’ ‘0100 1100’
atomics.w	TLX.vc.3, TLX.dcp.3	4



The AFU is requesting an atomic memory operation specified by the cmd_flag. For this request, operands are provided with the command and are found in memory as specified by the EA.

This command is specified with immediate data. Credits for both the VC and DCP shall be obtained before this command is serviced by the TLX.

The AFU TLX shall not service this command unless all data specified by pLength (pL) is available to be sent.

- The dot-n form indicates that the results of the address translation may not be installed into the host’s ATC as part of ATC miss handling.

There shall be a single response to this command. The host shall respond with either a **write_response** or a **write_failed** response packet. The **write_failed** response indicates that the operation failed. See the response packet description for additional details.

Operation:

The command’s address shall be naturally aligned based on the operand length. The operand length, as specified by the pLength (pL) field, is restricted to 4- and 8-byte operands. That is, the pLength shall be {‘010’, ‘011’}. All other values of pLength are reserved.

The number of operands specified by this command is determined by an examination of the cmd_flag. Two or three operands may be specified as shown in *Table 2-6* on page 60. The operands are designated as “A,” “A2,” and “V”. The command’s address specifies the location of each operand as follows:

- Operand A is located in memory at the address specified by the EA and shall be naturally aligned. When the cmd_flag indicates the use of operand A2, the address of operand A is further constrained and shall not target locations at 32n-$2^{\text{bin2dec(‘pL’)}}$, where n = 1, 2, 3...
- Operand A2 is located in memory at the address specified by the EA plus an offset specified by the width of the operands.

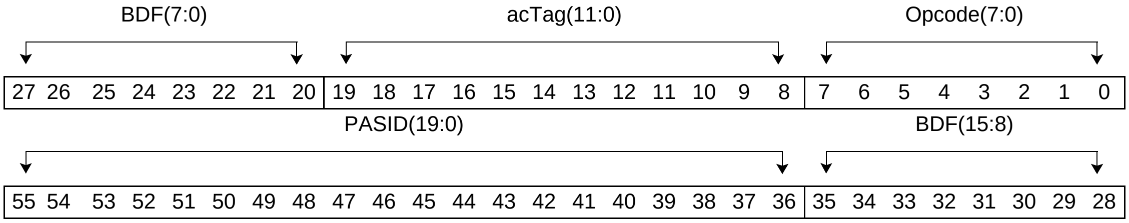
- Operand V is provided as the command's write data. Operand V shall be aligned within one of the following:
  - a 64-byte data flit based on address bits 5:0 specified by the command
  - a 32-byte data field carried in a control flit based on address bits 4:0 specified by the command
  - an 8-byte data field carried in a control flit based on address 2:0 specified by the command

The endianness of the operands is specified by the E bit. The value of E might not affect the result of the operation specified by the cmd_flag. This is noted in the operation description found in *Table 2-6*.

Table 2-6. The cmd_flag specification for ***amo_w***

cmd_flag	Operation name and description
'0000'	**Store and Add** Operands are unsigned integers. $A \leftarrow A + V$ Overflow conditions are not reported.
'0001'	**Store and XOR** Operands are bit vectors. E has no effect on the operation. $A \leftarrow V \oplus A$
'0010'	**Store and OR** Operands are bit vectors. E has no effect on the operation. $A \leftarrow V \vee A$
'0011'	**Store and AND** Operands are bit vectors. E has no effect on the operation. $A \leftarrow V \wedge A$
'0100'	**Store and maximum unsigned.** Operands are unsigned integers. $A \leftarrow \text{Max}(A, V)$ A is unmodified when A is greater than or equal to V.
'0101'	**Store and maximum signed** Operands are signed two's complement integers. $A \leftarrow \text{Max}(A, V)$ A is unmodified when A is greater than or equal to V.
'0110'	**Store and minimum unsigned** Operands are unsigned integers. $A \leftarrow \text{Min}(A, V)$ A is unmodified when A is less than or equal to V.
'0111'	**Store and minimum signed** Operands are signed two's complement integers. $A \leftarrow \text{Min}(A, V)$ A is unmodified when A is less than or equal to V.
'1000 through '1011'	Reserved.
'1100'	**Store and compare twin** Operands are bit vectors. E has no effect on the operation. When A = A2, then ($A \leftarrow V$, $A2 \leftarrow V$)
'1101' through '1111'	Reserved.

acTag Assignment	**assign_actag**	‘0101 0000’
acTag mgmt	TLX.vc.3	2



This command is used by the attached OpenCAPI device to assign an acTag value to a BDF and PASID combination. The OpenCAPI device uses this command to manage the contents of the acTag table. See *Section 4 The acTag table* on page 101 for the use of this command, the acTag table, and the management requirements placed on the OpenCAPI device.

This command is serviced when it reaches the head of the VC in the TL. It is not added to a *service queue*.

This command is posted. No response is sent for this command.

Interrupt Request	**intrp_req**	‘0101 1000’
message	TLX.vc.3	4

stream_id(3:0)	acTag(11:0)	cmd_flag(3:0)	Opcode(7:0)
27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0

Obj_handle(27:0)
55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28

Obj_handle(55:28)
83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56

Reserved	AFUTag(15:0)	Obj_handle(63:56)
111 110 109 108	107 106 105 104 103 102 101 100 99 98 97 96 95 94 93 92	91 90 89 88 87 86 85 84

This command is used to request interrupt service on the host. No data is transfered with this request. The specification of the object handle and the cmd_flag is found in the host’s platform architecture.

The response to this command is **intrp_resp**.

Engineering Note

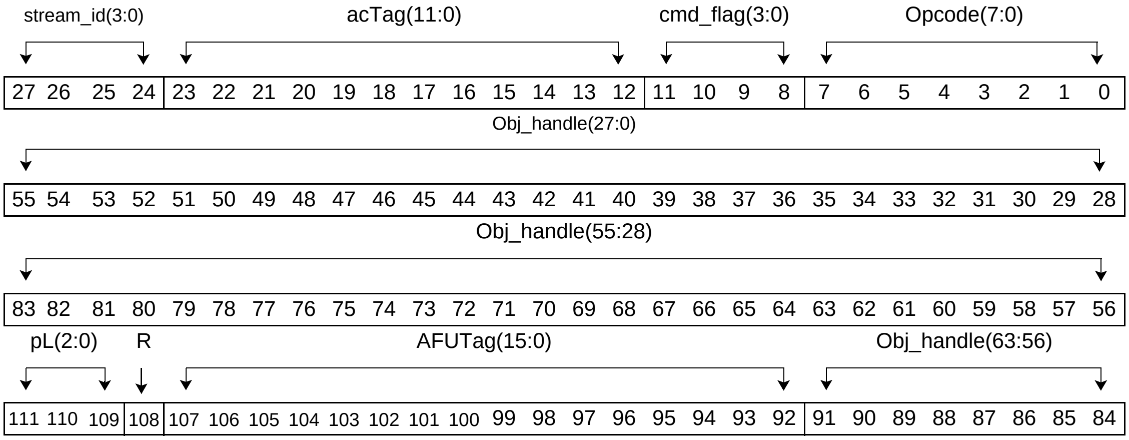
The AFUTag is passed back to the TLX in a response packet and has no control function in the TL as described in *Table 2-1 TL and TLX command operands* on page 31. The **intrp_resp**’s response code specification of intrp_pending indicates to the TLX that a subsequent **intrp_rdy** command is sent when the host is ready to service the interrupt. The **intrp_rdy** command contains the AFUTag that is specified in the original **intrp_req** command sent. While there are no requirements placed on the AFU to reserve the AFUTag used by the **intrp_req** command until the **intrp_rdy** command is received by the TLX, it is strongly recommended that an AFU implementation do so. It is an AFU implementation choice to use the AFUTag to precisely determine which interrupt to retry.

Developer Note

Both the command flag and the object handle fields specified for this command are specified by the host’s platform architecture.

The attached OpenCAPI device provides MMIO space where the combination of the object handle and the command flag associated with this command are located. The manufacturer of the OpenCAPI device determines the number of command-flag and object-handle combination entries supported based on what is supported by the host and the function provided by the device.

Interrupt Request	**intrp_req.d**	‘0101 1010’
message	TLX.vc.3, TLX.dcp.3	4



This command is used to request interrupt service on the host. Data, with the length specified by the pLength (pL) field, is transfered with this request. The data shall be sent in a 64-byte data flit.The alignment of the data within the data flit is specified by the host’s platform architecture. The specification of the object handle and the command flag is found in the host’s platform architecture.

The response to this command is **intrp_resp**.

Engineering Note

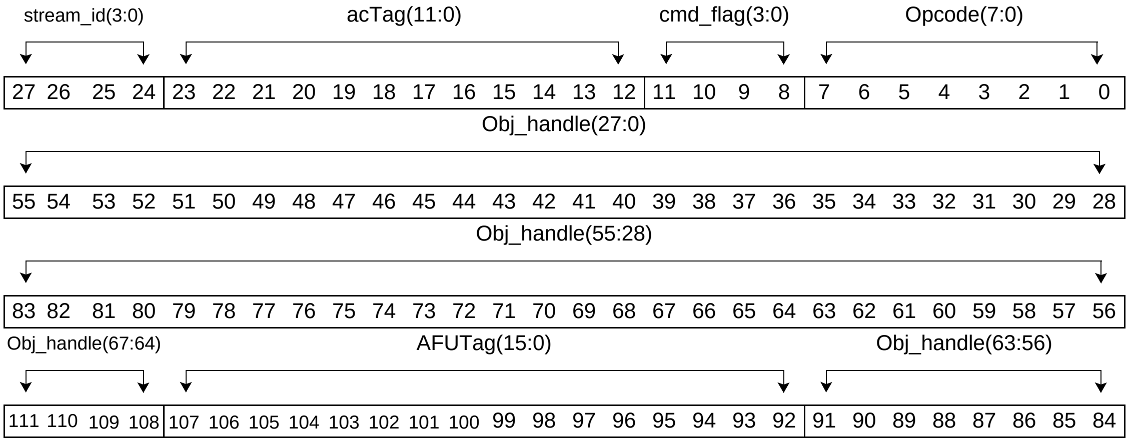
The AFUTag is passed back to the TLX in a response packet and has no control function in the TL as described in *Table 2-1* on page 31. The **intrp_resp**’s response code specification of intrp_pending indicates to the TLX that a subsequent **intrp_rdy** command is sent when the host is ready to service the interrupt. The **intrp_rdy** command contains the AFUTag that is specified in the original **intrp_req.d** command sent. While there are no requirements placed on the AFU to reserve the AFUTag used by the **intrp_req.d** command until the **intrp_rdy** command is received by the TLX, it is strongly recommended that an AFU implementation do so. It is an AFU implementation choice to use the AFUTag to precisely determine which interrupt to retry.

Developer Note

The command flag, data, and the object handle fields specified for this command are specified by the host’s platform architecture.

The attached OpenCAPI device provides MMIO space where the combination of the object handle, data, and the command flag associated with this command are located. The manufacturer of the OpenCAPI device determines the number of command-flag and object-handle combination entries supported based on what is supported by the host and the function provided by the device.

Wake host thread	**wake_host_thread**	‘0101 1100’
message	TLX.vc.3	4



This command is used to wake a thread on the host. The specification of the object handle and the command flag is found in the host’s platform architecture.

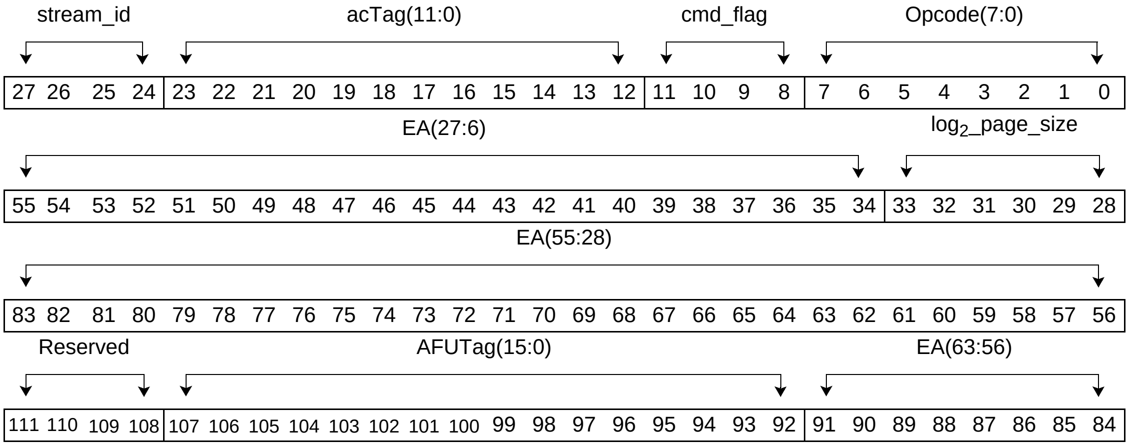
Results are returned to the AFU using **wake_host_resp**.

Developer Note

See the Developer note found in the description of **intrp_req** for details on the specification of the object handle and command flag and the requirements this specification places on the OpenCAPI device.

wake_host_resp indicates if the operation was successful, or if an interrupt is required.

Address translation prefetch	**xlate_touch** **xlate_touch.n**	‘0111 1000’ 0111 1100’
address translation managment	TLX.vc.3	4



This command is used to request address translation prefetch for the address (EA) specified. The address can specify any 64-byte aligned address (EA). The $\log_2$_page_size field specifies the page size for an age-out ATC entry request and is reserved for an address translation request. See the specification of the command flag, bit 0, in *Table 2-7*.

- The dot-n form indicates that the results of the address translation may not be installed into the host’s ATC as part of ATC miss handling.

Table 2-7 provides the specification of the cmd_flag field. *Figure 2-1* on page 67 provides the architectural description of the command’s operation.

Table 2-7. The cmd_flag specification for ***xlate_touch*** *(all forms)* (Page 1 of 2)

cmd_flag bit	Description
3	Reserved.
2	0 Light-weight touch (lwt). Address translation stops and returns status if software intervention is required to complete the address translation request. Software intervention shall not be initiated. 1 Heavy-weight touch (hwt). Address translation invokes software intervention if required to complete the address translation request. Status is returned immediately. The result of the software intervention is reported to the AFU using **xlate_done**.
1	0 Read-only access requested (ro). Read permission is requested by this address translation request. Write permission may be obtained. 1 Write access requested (w). Write permission is requested by this address translation request.

Table 2-7. The cmd_flag specification for ***xlate_touch*** *(all forms)* (Page 2 of 2)

<table>
<tr><th>cmd_flag bit</th><th>Description</th></tr>
<tr><td>0</td><td>0 Address translation request (xlate). log$_2$_page_size is reserved.
1 Age-out ATC entry request (age_out). log$_2$_page_size specifies the page size of the entry to be aged out.

<b>Engineering note</b>
<b>xlate_touch</b> can be used to update the LRU mechanism of the host’s ATC. cmd_flag(0) can be used to provide hints to the host.
0 Address translation is invoked. If an entry is found in the ATC, or one is added, that entry is marked MRU
1 Address translation is invoked. If an entry is found in the ATC, that entry is marked as LRU.
It is determined by the host implementation if early aging causes immediate invalidation of the matching ATC entries, the entries are marked as LRU, or no action is taken. A host implementation may chose to ignore the LRU hints described above.
The page size associated with an EA is provided in the <b>touch_resp</b> of a previous <b>xlate_touch</b>. The OpenCAPI device shall retain this information when making an age-out request.</td></tr>
<tr><td colspan="2">cmd_flag encode specification:
0000 xlate, lwt.ro
0001 age out
0010 xlate, lwt_w
0011 Reserved
0100 xlate, hwt.ro
0101 Reserved
0110 xlate, hwt.w
0111 Reserved
The use of reserved code points results in fatal errors. See <i>Table 7-1 Error event specification</i> on page 117.
The use of a dot-n form and age out is an error. See <i>Age out specified for</i> <b><i>xlate_touch.n on page 118</i></b> for details.</td></tr>
</table>

Developer Note

Figure 2-1 on page 67 does not show validation of the addressContext or the impacts of other hardware-driven events that might terminate this operation. See the specification of **touch_resp** for details of the result specification.

Figure 2-1. Address translation sequence: ***xlate_touch*** *(Page 1 of 3)*

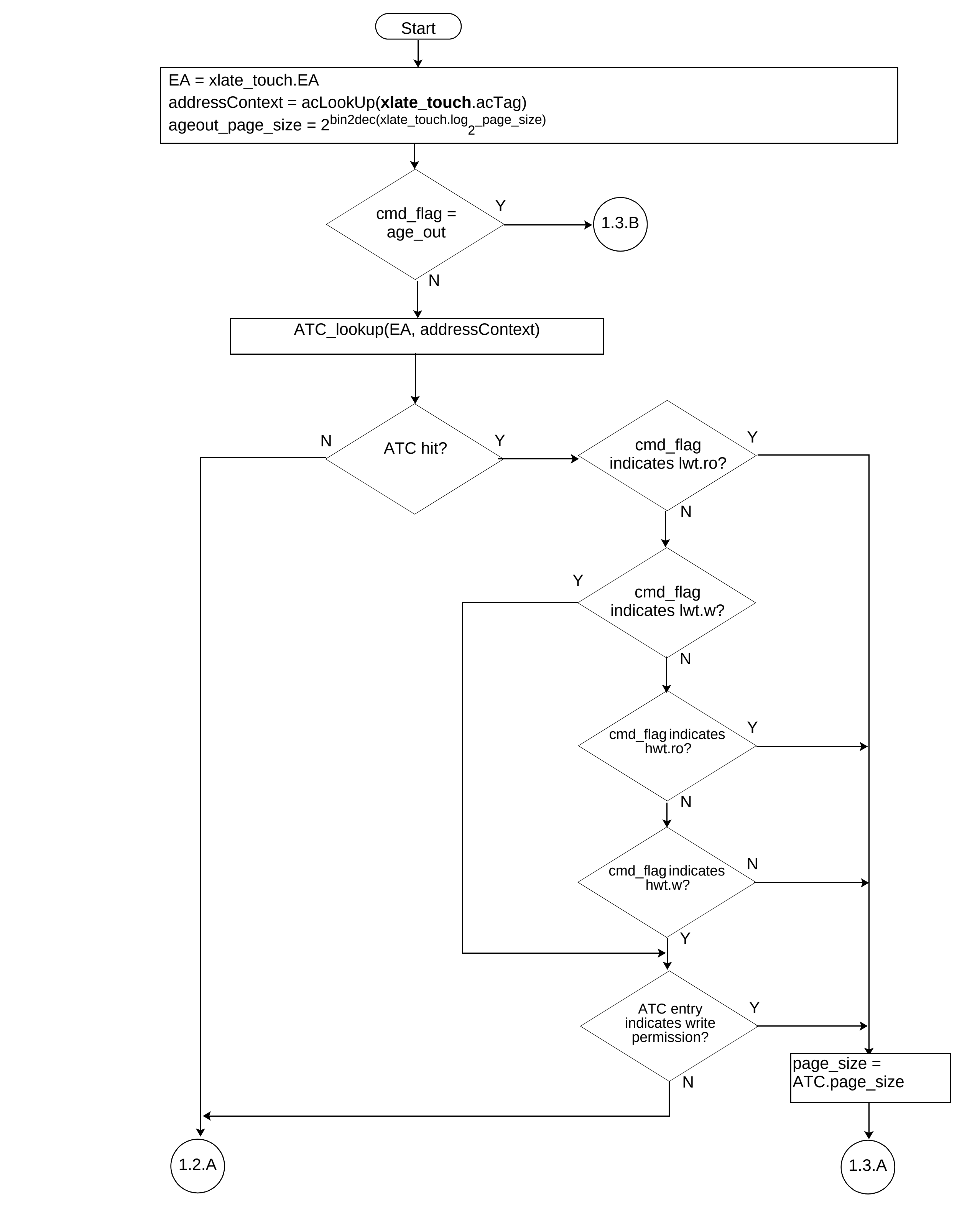

Start
EA = xlate_touch.EA
addressContext = acLookUp(xlate_touch.acTag)
ageout_page_size = 2^bin2dec(xlate_touch.log2_page_size)
cmd_flag = age_out
Y
N
1.3.B
ATC_lookup(EA, addressContext)
ATC hit?
N
Y
cmd_flag indicates lwt.ro?
Y
N
cmd_flag indicates lwt.w?
Y
N
cmd_flag indicates hwt.ro?
Y
N
cmd_flag indicates hwt.w?
N
Y
ATC entry indicates write permission?
Y
N
page_size = ATC.page_size
1.2.A
1.3.A

Figure 2-1. Address translation sequence: ***xlate_touch*** *(Page 2 of 3)*

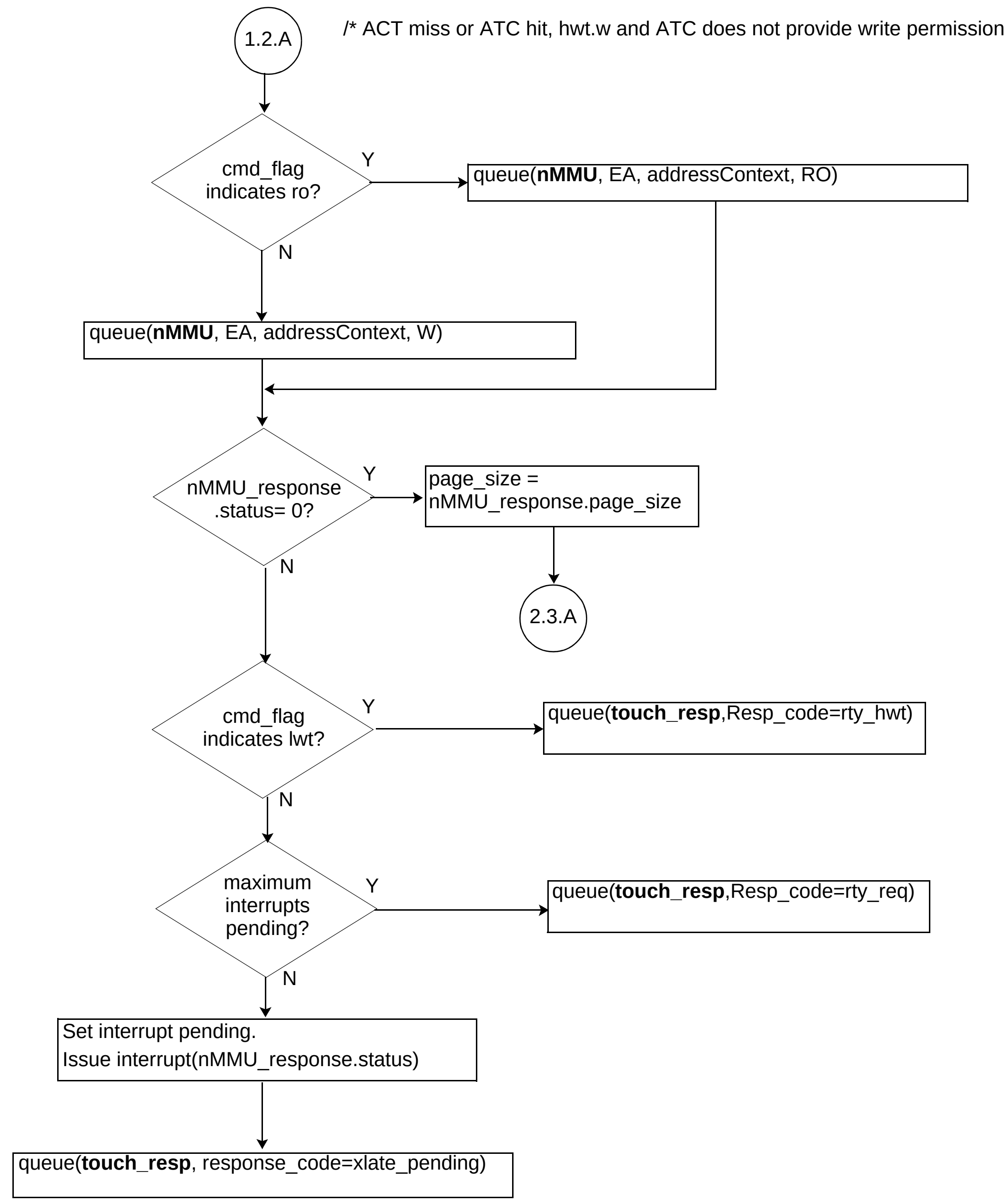
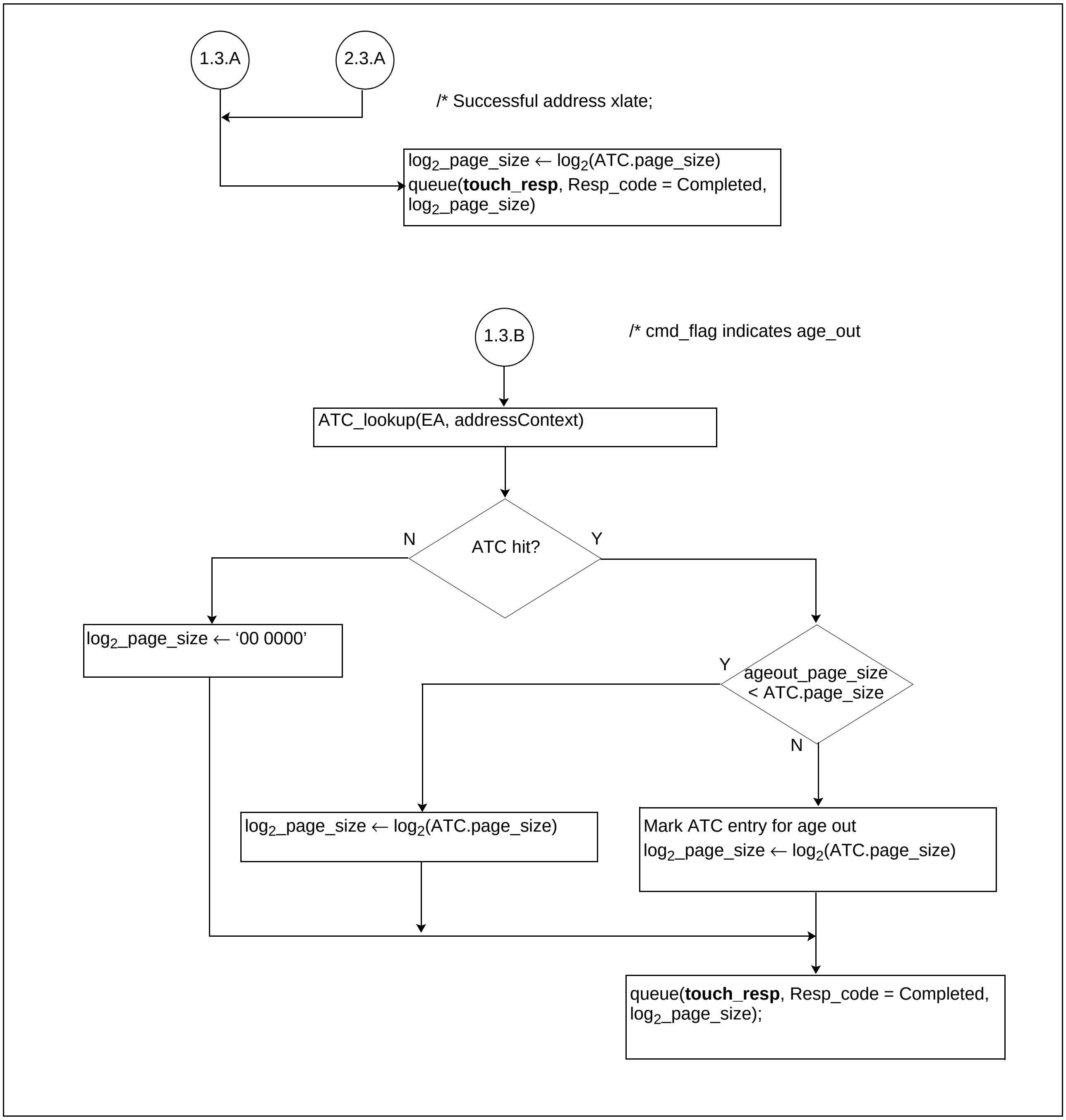

1.2.A
/* ACT miss or ATC hit, hwt.w and ATC does not provide write permission
cmd_flag indicates ro?
Y
queue(**nMMU**, EA, addressContext, RO)
N
queue(**nMMU**, EA, addressContext, W)
nMMU_response .status= 0?
Y
page_size = nMMU_response.page_size
2.3.A
N
cmd_flag indicates lwt?
Y
queue(**touch_resp**,Resp_code=rty_hwt)
N
maximum interrupts pending?
Y
queue(**touch_resp**,Resp_code=rty_req)
N
Set interrupt pending.
Issue interrupt(nMMU_response.status)
queue(**touch_resp**, response_code=xlate_pending)

Figure 2-1. Address translation sequence: ***xlate_touch*** *(Page 3 of 3)*


1.3.A
2.3.A
/* Successful address xlate;
log2_page_size ← log2(ATC.page_size)
queue(touch_resp, Resp_code = Completed, log2_page_size)
1.3.B
/* cmd_flag indicates age_out
ATC_lookup(EA, addressContext)
ATC hit?
N
Y
log2_page_size ← '00 0000'
ageout_page_size < ATC.page_size
Y
N
log2_page_size ← log2(ATC.page_size)
Mark ATC entry for age out
log2_page_size ← log2(ATC.page_size)
queue(touch_resp, Resp_code = Completed, log2_page_size);

Results are returned to the AFU using **touch_resp**.

Engineering Note

The AFUTag is passed back to the TLX in a response packet and has no control function in the TL as described in *Table 2-1* on page 31. The **touch_resp**'s response code specification of xlate_pending indicates to the TLX that a subsequent **xlate_done** command is sent when the host is ready to service the translation request.

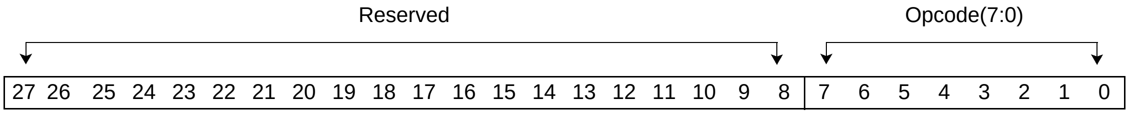
- The **xlate_done** command contains the AFUTag that is specified in the original **xlate_touch** command sent. While there are no requirements placed on the AFU to reserve the AFUTag used by the **xlate_touch** command until the **xlate_done** command is received by the TLX, it is strongly recommended that an AFU implementation do so. It is an AFU implementation choice to use the AFUTag to precisely determine which address translation to retry. The alternative is likely to be less efficient.
- **xlate_done** uses TL.vc.0. The implementation shall ensure that the **touch_resp** carrying the response code of xlate_pendng is added to the VC prior to the **xlate_done**.

2.4 TL CAPP response packets

TL responses are sent from the host to the AFU. An alphabetical list of the TL responses follows; each response is hyperlinked to its specification. In this section, the TL response specifications are in opcode order.

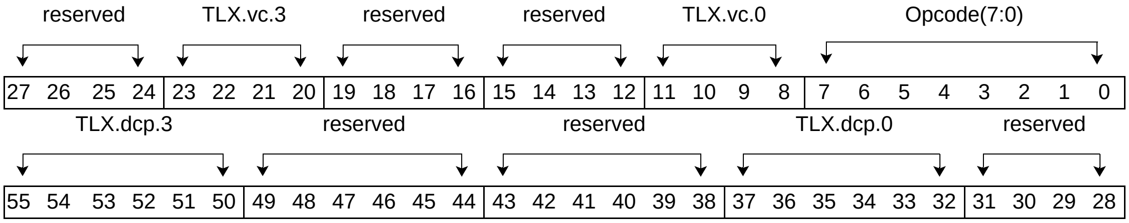
intrp_resp	**nop**	**read_failed**	**read_response**
read_response.ow	**read_response.xw**	**return_tlx_credits**	**touch_resp**
wake_host_resp	**write_response**	**write_failed**	

No operation	**nop**	‘0000 0000’
NA	NA	1



This response has no operands and performs no action. It is discarded at the TLX.

Return TLX credits	**return_tlx_credits**	‘0000 0001’
credit return	NA	2



This response packet is used by the TL to return VC and DCP credits to the TLX. There is no VC associated with this response, and credits are not required to service this response. Each TLX.* field contains the number of credits being returned.

This response packet shall be placed only in slots 1 to 0 of any control flit using a template which specifies those slots as a 2-slot or larger location.

TLX.vc.{0, 3} and TLX.dcp.{0, 3} credits are returned. TLX credits are for resources owned by the TL that the TLX consumes. The TL controls the total number of credits for each of the VC and DCP it provisions the TLX with.

touch response	**touch_resp**	‘0000 0010’
address translation management	TL.vc.0	2

Reserved	AFUTag(15:0)	Opcode(7:0)
27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

Resp_code(3:0)	Reserved	$\log_2$_page_size
55 54 53 52	51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34	33 32 31 30 29 28

This is a response to an **xlate_touch** command. $\log_2$_page_size is valid only when the Resp_code = Completed, and when xlate_touch specifies address translation (xlate), otherwise the field is reserved. The Resp_code field is specified in *Table 2-8*.

*Table 2-8. The Resp_code specification for **touch_resp***

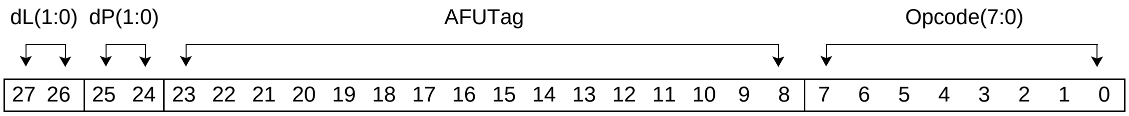
Resp_code encode	Description
‘0000’	Completed. Address translation completed successfully.
‘0001’	Retry using the heavy-weight touch specification (rty_hwt). The translation could not be completed using the light-weight touch (lwt) specified by the **xlate_touch** command.
‘0010’	Retry request (rty_req). Indicates that the address translation could not be completed at this time. An address translation attempt may be made a later time. This is a long *back-off event*.
‘0011’	Reserved.
‘0100’	Translation pending (xlate_pending). Indicates that the address translation could not be completed. The ATC did not contain the translation, and software was invoked. An asynchronous **xlate_done** TL command shall be sent when software actions have completed. It is strongly recommended that the device wait for **xlate_done** to be received before retrying the operation. However, using a retry back off mechanism is permitted to determine when to retry the command. Such an implementation shall examine **xlate_done** for the results of the address translation and take action based on those results. • **xlate_done** uses TL.vc.0. The implementation shall ensure that the **touch_resp** carrying the response code of xlate_pendng is added to the VC prior to the **xlate_done**.
‘0101’ - ‘1011’	Reserved.
‘1100’	Reserved.
‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be recovered. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the host’s platform architecture.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

touch_resp responds to the TLX commands found in *Table 2-9*. For each command only the Resp_codes indicated with a Y may be used. Resp_code with N shall not be used.

Table 2-9. ***touch_resp*** *Resp_code use by TLX command*

TLX command	Completed (0)	rty_hwt (1)	rty_req (2)	xlate_pending (4)	Failed (14)
xlate_touch	Y	Y	Y	Y	Y
xlate_touch.n	Y	Y	Y	Y	Y

Read response	**read_response**	‘0000 0100’
Read data return	TL.vc.0, TL.dcp.0	1



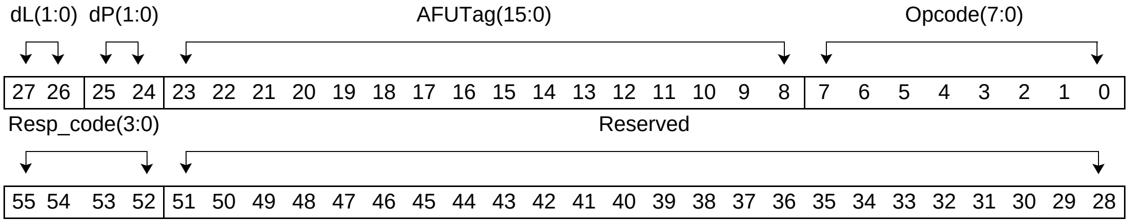
In response to a non-cacheable read command initiated by the AFU, the host is returning data. The data may be returned in a data flit, or may be returned in multiple 32-byte data fields carried in control flits. The AFU can determine which command to associate the data with by using the AFUTag provided with the command and returned with the response.

The dLength (dL) field indicates the amount of data contained in this response. Multiple read response packets may be received for a single read command. When the dLength field in the response does not match the full amount of data requested by the command, the dPart (dP) field is used to indicate the offset within the *naturally aligned data block* specified by the address in the read command. For multiple responses to a single command, the AFUTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TLX may see the values of dPart returned in any order. When multiple responses are received for a read command, a combination of **read_response**, **read_response.ow**, and **read_failed** responses may be received. Taken together, all responses shall contain a combination of dPart and implied length or dLength to cover the command's dLength specification.

The dLength and dPart fields shall be specified as 64 bytes and offset at 0 for **pr_rd_wnitc** and any of the commands classified as mem_atomics that return data. A single response covers the entire operation. Data is aligned within the data flit based on the command’s address bits 5:0.

This response is specified with immediate data. Credits for both the VC and DCP shall be obtained before this response is serviced by the TL.

Read failed response	**read_failed**	‘0000 0101’
Read data return	TL.vc.0	2



In response to a read command initiated by the AFU, the host is indicating that the read failed. The AFU determines which command to associate the failure with by using the AFUTag provided with the command and returned with the response.

The dLength and dPart fields specify how much of the read operation is being reported. Multiple read response packets may be received for a single read command. When the dLength field in the response does not match the full amount of data requested by the command, the dPart field is used to indicate the offset within the *naturally aligned data block* specified by the address in the read command. For multiple responses to a single command, the AFUTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TLX may see the values of dPart returned in any order.

- When multiple responses are received for **rd_wnitc**, a combination of **read_response**, **read_response.ow**, and **read_failed** responses may be received. Taken together, all responses shall contain a combination of dPart and implied length or dLength to cover the command's dLength specification.

The dLength and dPart fields shall be specified as 64 bytes and offset at 0 for **pr_rd_wnitc**, **amo_rd**, **amo_rw**, and all dot variants of these commands. A single response shall be returned for these commands.

The Resp_code field indicates the type of failure being reported. The Resp_code field is specified in *Table 2-10*.

*Table 2-10. The Resp_code specification for **read_failed*** (Page 1 of 2)

Resp_code encode	Description
‘0000’ - ‘0001’	Reserved.
‘0010’	Retry request (rty_req). Use of this code point might be due to an event in the host that may require software intervention, or may indicate that address translation could not be completed for the command at this time, or may be due to a hardware recovery mechanism. The operation may be retried by the device. This is a long back-off event.
‘0011’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

Table 2-10. The Resp_code specification for ***read_failed*** (Page 2 of 2)

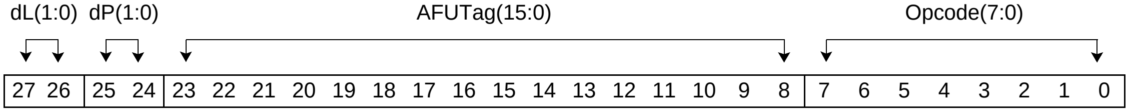
Resp_code encode	Description
‘0100’	Translation pending (xlate_pending). Indicates that the address translation could not be completed. The ATC did not contain the translation, and software was invoked. An asynchronous **xlate_done** TL command shall be sent when software actions have completed. It is strongly recommended that an AFU wait for the **xlate_done** before retrying the command. However, using a retry back off timer is permitted to determine when to retry the command. Such an implementation shall examine **xlate_done** for the results of the address translation and take action based on those results. • **xlate_done** uses TL.vc.0. The implementation shall ensure that the **read_failed** carrying the response code of xlate_pendng is added to the VC prior to the **xlate_done**.
‘0101’	Reserved
‘0110’	Reserved.
‘0111’	Reserved.
‘1000’	Data error (dError). The host’s protocol stack operation has completed.The data obtained by the host has been corrupted and is not correctable. This may be a recoverable error by retrying the operation. See the device documentation and *Section 2.1.1* for additional information. **Engineering note** A dError condition may also be reported using **read_response**, **read_response.ow**, or **read_response.xw**, as appropriate for the TLX command, and shall indicate that the data is bad using the bad data indication in the control flit as specified for the data carrier used.
‘1001’	Unsupported operand length. The operation specifies an operand length that is not supported by the device. A retry of the operation shall not be successful.
‘1010’	Reserved.
‘1011’	Bad address specification. The address specified by the command is not naturally aligned on a boundary specified by the operand length. Additional restrictions for address specification are specified in the operation descriptions of the TLX command **amo_rd** *on page 55*. A retry of the operation shall not be successful.
‘1100’	Reserved.
‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be recovered. This code point indicates that the state of the host due to the error occurrence does not allow a successful retry of the operation. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the host’s platform architecture.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

read_failed responds to the TLX commands found in *Table 2-11*. For each command only the Resp_codes indicated with a Y may be used. Resp_code indicated with an N shall not be used.

*Table 2-11. **read_failed** Resp_code use by TLX command*

TLX command	rty_req (2)	xlate_pending (4)	dError (8)	Unsupported operand length (9)	Bad address specification (11)	Failed (14)
rd_wnitc	Y	Y	Y	N	N	Y
pr_rd_wnitc	Y	Y	Y	N	Y	Y
rd_wnitc.n	Y	Y	Y	N	N	Y
pr_rd_wnitc	Y	Y	Y	N	Y	Y
amo_rd	Y	Y	Y	N	Y	Y
amo_rd.n	Y	Y	Y	N	Y	Y
amo_rw	Y	Y	Y	N	Y	Y
amo_rw.n	Y	Y	Y	N	Y	Y

Write response	**write_response**	‘0000 1000’
write response	TL.vc.0	1



This packet is used in response to a write command (that is, **dma_w**, **dma_w.be**, **dma_pr_w**, **amo_w**) operation that has succeeded. The AFU determines which command to associate with this response by using the AFUTag provided with the command and returned with the response. Data specified by this response is global visible. That is, a subsequent read shall see the new data.

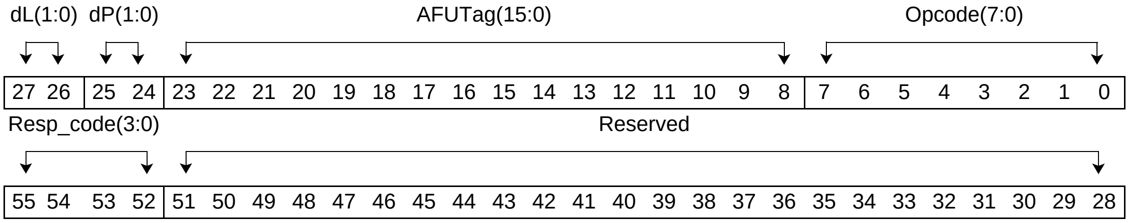
For **dma_w**, the dLength (dL) and dPart (dP) fields specify how much of the write operation is being reported. A single response may cover the entire operation. For a single response, the dLength must match the dLength specified by the command, and dPart must indicate a starting offset of 0. Multiple write response packets may be received for a single write command. When the dLength field in the response does not match the full amount of data requested by the command, the dPart field is used to indicate the offset from the starting address specified in the write command. For multiple responses to a single command, the AFUTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TLX may see the values of dPart returned in any order. When multiple responses are received for a write command, a combination of **write_response** and **write_failed** responses may be received. Taken together, all responses shall contain a combination of dLength and dPart to cover the command's dLength specification.

For **dma_pr_w** and **amo_w** commands, only one response is expected; dLength and dPart shall be specified as 64 bytes and offset at 0.

Engineering note

A **write_response** is used in response to a **dma_w** regardless of how the data was transported to the TLX. The response is on a multiple of 64-byte blocks even when the data might have been moved using 32-byte data fields in control flits. The AFU shall gather the completion of the 32-byte transfers and report the results as if the transfers occurred in 64-byte address-aligned data flits.

Write failed response	**write_failed**	‘0000 1001’
write response	TL.vc.0	2



In response to a write command initiated by the AFU, the host is indicating that the write failed. The AFU determines which command to associate the failure with by using the AFUTag provided with the command and returned with the response.

The dLength and dPart fields specify how much of the write operation is being reported. A single response may cover the entire operation. For a single response, the dLength must match the dLength specified by the command, and dPart must indicate a starting offset of 0. Multiple write response packets may be received for a single write command. When the dLength field in the response does not match the full amount of data requested by the command, the dPart field is used to indicate the offset from the starting address specified in the write command. For multiple responses to a single command, the AFUTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TLX may see the values of dPart returned in any order. When multiple responses are received for a write command, a combination of **write_response** and **write_failed** responses may be received. Taken together, all responses shall contain a combination of dLength and dPart to cover the command's dLength specification.

This response shall be returned when the operation fails.

For **dma_w.be**, **dma_pr_w**, **amo_w,** and all dot variants of these commands, only one response shall be returned; dLength and dPart shall be specified as 64 bytes and offset at 0.

The Resp_code field indicates the type of failure being reported. The Resp_code field is specified in *Table 2-12*.

*Table 2-12. The Resp_code specification of **write_failed*** (Page 1 of 2)

Resp_code encode	Description
‘0000’ - ‘0001’	Reserved.
‘0010’	Retry request (rty_req). Use of this code point might be due to an event in the host that may require software intervention, or may indicate that address translation could not be completed at this time, or may be due to a hardware recovery mechanism. The operation may be retried by the device. This is a long back-off event.
‘0011’	Reserved.
‘0100’	Translation pending (xlate_pending). Indicates that the address translation could not be completed. The ATC did not contain the translation, and software was invoked. An asynchronous **xlate_done** TL command shall be sent when software actions have completed. It is strongly recommended that an AFU wait for the **xlate_done** before retrying the command. However, using a retry back off timer is permitted to determine when to retry the command. Such an implementation shall examine **xlate_done** for the results of the address translation and take action based on those results. • **xlate_done** uses TL.vc.0. The implementation shall ensure that the **write_failed** carrying the response code of xlate_pendng is added to the VC prior to the **xlate_done**.
‘0101’ - ‘0110’	Reserved.
‘0111’	Reserved.
‘1000’	Data error (dError). The host’s protocol stack operation completed. The received data was UE data, or might have been marked bad in the control flit associated with the data transfer, or might have been damaged in the host. Changes, if any, to the memory location specified by the response are globally visible. The memory location shall contain SUE data. **Engineering note** If an implementation is unable to modify the memory location specified by the command to contain SUE data, the implementation shall not report a dError. The implementation shall report a Failed. **Engineering note** A dError condition may also be reported by the consumer of the data. That is, the reporting of the dError condition may be delayed until the data is consumed by a read operation. This requires that the actions taken when the error condition is detected either shall cause the memory location to contain SUE data or shall use an alternate method to report the data is invalid prior to or when it is consumed. When either of these methods are used, the host may response with **write_response** instead of a **write_failed**.
‘1001’	Unsupported operand length. The operation specifies an operand length that is not supported by the device. A retry of the operation shall not be successful.
‘1010’	Reserved.
‘1011’	Bad address specification. The address specified is not naturally aligned on a boundary specified by the operand length. A retry of the operation shall not be successful.
‘1100’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

Table 2-12. The Resp_code specification of ***write_failed*** (Page 2 of 2)

Resp_code encode	Description
‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be recovered. This code point indicates that the state of the host due to the error occurrence does not allow a successful retry of the operation. This includes the following: • A dError event was detected and the implementation is unable to modify the memory location specified by the command to contain SUE data. Changes, if any to the memory location specified by the response are globally visible. The memory location may be unmodified, or may contain undefined data. • Any other failure detected by the host that is not included in any of the specified response codes. The failure may cause the modification of the memory location specified by the command. Changes, if any to the memory location specified by the response are globally visible. The memory location may be unmodified, may contain undefined data, or may contain SUE data. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the host’s platform architecture.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

write_failed responds to the TLX commands found in *Table 2-13*. For each command only the Resp_codes indicated with a Y may be used. Resp_code indicated with an N shall not be used.

Table 2-13. **write_failed** *Resp_code use by TLX command*

TLX command	rty_req (2)	xlate_pending (4)	dError (8)	Unsupported operand length (9)	Bad address specification (11)	Failed (14)
dma_w	Y	Y	Y	N	Y	Y
dma_w.n	Y	Y	Y	N	Y	Y
dma_w.be	Y	Y	Y	N	N	Y
dma_w.be.n	Y	Y	Y	N	N	Y
dma_pr_w	Y	Y	Y	N	Y	Y
dma_pr_w.n	Y	Y	Y	N	Y	Y
amo_w	Y	Y	Y	Y	Y	Y
amo_w.n	Y	Y	Y	Y	Y	Y

Interrupt response	**intrp_resp**	‘0000 1100’
message response	TL.vc.0	2

Reserved	AFUTag(15:0)	Opcode(7:0)
27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

Resp_code(3:0)	Reserved
55 54 53 52	51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28

This packet is used in response to **intrp_req**, and **intrp_req.d** commands.

The response code indicates that the interrupt was successfully initiated or provides error status. The Resp_code field is specified in *Table 2-14*.

*Table 2-14. The Resp_code specification for **intrp_resp***

Resp_code encode	Description (Page 1 of 2)
‘0000’	Interrupt request accepted.
‘0001’	Reserved.
‘0010’	Retry request (rty_req). Use of this code point might be due to an event in the host that may require software intervention, or may indicate that address translation could not be completed at this time, or may be due to a hardware recovery mechanism. The operation may be retried by the device. This is a long back-off event.
‘0011’	Reserved.
‘0100’	Interrupt resources pending (intrp_pending). Indicates that the operation could not be completed at this time requiring additional software intervention. Software intervention has been successfully invoked. An asynchronous **intrp_rdy** TL command shall be sent when software actions have completed and the operation can be retried. It is strongly recommended that an AFU wait for the **intrp_rdy** before retrying the command. However, using a retry back off timer is permitted to determine when to retry the command. Such an implementation shall examine **intrp_rdy** for the results and take action based on those results. • **intrp_rdy** uses TL.vc.0. The implementation shall ensure that the **intrp_resp** carrying the response code of intrp_pending is added to the VC prior to the **intrp_rdy**.
‘0101’ - ‘0110’	Reserved.
‘0111’	Reserved.
‘1000’	Data error (dError). Used only in response to **intrp_req.d**. The received data was corrupted and not correctable, or might have been marked bad in the control flit associated with the data transfer, or might have been damaged in the host. The operation is aborted.
‘1001’	Unsupported operand length. Used only in response to **intrp_req.d**. The operation specifies an operand length that is not supported by the device. A retry of the operation shall not be successful.
‘1010’	Reserved.
‘1011’	Bad object handle specification. The object handle is specified by the platform architecture. A retry of the operation shall not be successful.
‘1100’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

Table 2-14. The Resp_code specification for ***intrp_resp***

Resp_code encode	Description (Page 2 of 2)
‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be recovered.This code point indicates that the state of the host due to the error occurrence does not allow a successful retry of the operation. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the host’s platform architecture.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

intrp_resp responds to the TLX commands found in *Table 2-15*. For each command only the Resp_codes indicated with a Y may be used. Resp_code indicated with an N shall not be used.

Table 2-15. **intrp_resp** *Resp_code use by TLX command*

TLX command	rty_req (2)	xlate_pending (4)	dError (8)	Unsupported operand length (9)	Bad address specification (11)	Failed (14)
intrp_req	Y	Y	N	N	Y	Y
intrp_req.d	Y	Y	Y	Y	Y	Y

Read response	**read_response.ow**	‘0000 1101’
Read data return	TL.vc.0, TL.dcp.0	1

R	dP(2:0)			AFUTag																Opcode(7:0)							
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

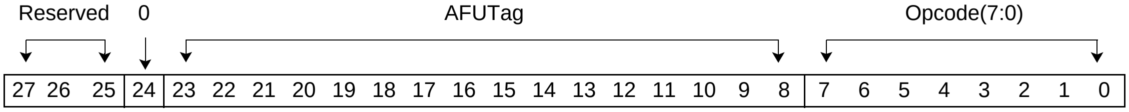
In response to a non-cacheable read command initiated by the AFU, the host is returning data using a 32-byte data field carried in control flits. The AFU can determine which command to associate the data with by using the AFUTag provided with the command and returned with the response.

The response implies a data length of 32 bytes. Multiple read response packets may be received for a single read command. The dPart (dP) indicates the offset within the *naturally aligned data block* specified by the address in the read command. For multiple responses to a single command, the AFUTag is unchanged. That is, the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TLX may see the values of dPart returned in any order. When multiple responses are received for a read command, a combination of **read_response**, **read_response.ow**, and **read_failed** responses may be received. Taken together, all responses shall contain a combination of dPart and implied lengths or dLength to cover the command's dLength specification.

The dPart field shall be specified as offset at 0 for **pr_rd_wnitc** and any of the commands classified as mem_atomics that return data. A single response covers the entire operation. Data is aligned within the 32-byte data carrier based on the command’s address bits 4:0.

This response is specified with immediate data. Credits for both the VC and DCP shall be obtained before this response is serviced by the TL.

Read response	**read_response.xw**	‘0000 1110’
Read data return	TL.vc.0, TL.dcp.0	1

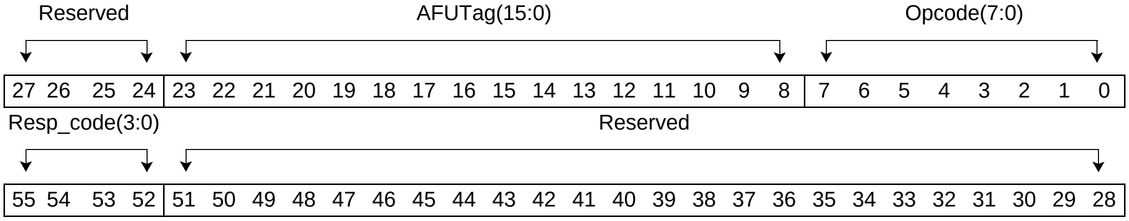


In response to a non-cacheable read command initiated by the AFU, the host is returning data using an 8-byte data field carried in a control flit. The AFU can determine which command to associate the data with by using the AFUTag provided with the command and returned with the response.

The response implies a data length of 8 bytes. The full amount of the data requested by the command is returned.

This response is specified with immediate data. Credits for both the VC and DCP shall be obtained before this response is serviced by the TL.

Wake Host Thread Response	**wake_host_resp**	‘0001 0000’
message response	TL.vc.0	2



This packet is used in response to a **wake_host_thread** command. The operation in the host was either successful in waking the thread specified by the command or it was not. The Resp_code field and the reporting priority when multiple errors are detected is specified in *Table 2-16*.

Table 2-16. The Resp_code specification for ***wake_host_resp*** (Page 1 of 2)

Resp_code encode	Description
‘0000’	Thread found. Thread woken.
‘0001’	Reserved.
‘0010’	Retry request (rty_req). Indicates that the operation could not be completed at this time. The operation may be retried at a later time. This is a long *back-off event*.
‘0011’	Reserved.
‘0100’	Reserved.
‘0101’	Thread not found. An interrupt is required to service the operation.
‘0110’	Reserved.
‘0111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

Table 2-16. The Resp_code specification for ***wake_host_resp*** (Page 2 of 2)

Resp_code encode	Description
‘1000’ - 1010’	Reserved.
‘1011’	Bad object handle specification. The object handle is specified by the platform architecture. A retry of the operation shall not be successful.
‘1100’	Reserved.
‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be recovered.This code point indicates that the state of the host due to the error occurrence does not allow a successful retry of the operation. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the host’s platform architecture.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

2.5 TLX AP response packets

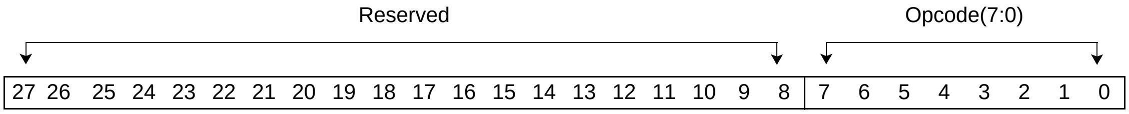
TLX responses are sent from the AFU to the host. An alphabetical list of the TLX responses follows; each response is hyperlinked to its specification. In this section, the TLX response specifications are in opcode order.

mem_cntl_done **mem_rd_fail** **mem_rd_response** **mem_rd_response.ow**

mem_rd_response.xw **mem_wr_fail** **mem_wr_response** **nop**

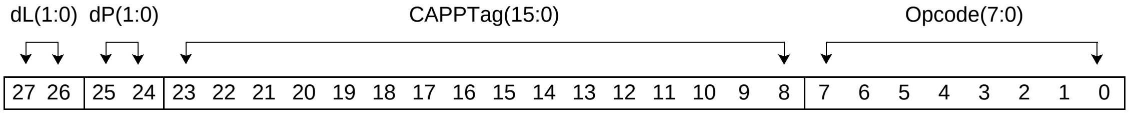
return_tl_credits

No operation	**nop**	‘0000 0000’
NA	NA	1



This response has no operands and performs no action. It is discarded at the TL.

Memory read response	**mem_rd_response**	‘0000 0001’
mem_response	TLX.vc.0, TLX.dcp.0	1



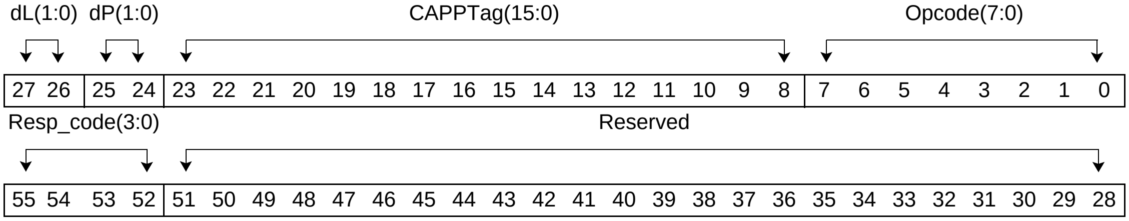
In response to a memory read command initiated by the host, the AFU is returning data. The data may be returned in a data flit, or may be returned in multiple 32-byte data carriers. The host can determine which command to associate the data with by using the CAPPTag provided with the command and returned with the response.

The dLength (dL) field indicates the amount of data contained in this response. Multiple response packets may be received for a single memory read command. When the dLength field in the response does not match the full amount of data specified by the command, the dPart (dP) field is used to indicate the offset within the *naturally aligned data block* specified by the address in the memory read command. For multiple responses to a single command, the CAPPTag is unchanged. That is, only the dLength and dPart fields may vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TL may see the values of dPart returned in any order. When multiple responses are received for a memory read command, a combination of **mem_rd_response**, **mem_rd_response.ow**, and **mem_rd_fail** responses may be received. Taken together, all responses shall contain a combination of dPart and implied length or dLength to cover the command's dLength specification.

For **pr_rd_mem** and **config_read**, the dLength and dPart fields shall be specified as 64 bytes and offset at 0. A single response shall be returned.

This response is specified with immediate data. Credits for both the VC and DCP shall be obtained before this response is serviced by the TLX.

Memory read failure	**mem_rd_fail**	‘0000 0010’
mem_response	TLX.vc.0	2



In response to a memory read command initiated by the host, the AFU is indicating that the read failed. The host determines which command to associate with the failure by using the CAPPTag provided with the command and returned with the response.

For **rd_mem**, the dLength (dL) and dPart (dP) fields specify how much of the read operation is being reported. A single response may cover the entire operation. For a single response, the dLength field must match the dLength specified by the command, and the dPart field must indicate a starting offset of 0. Multiple response packets may be received for a single memory read command. When the dLength field in the response does not match the full amount of data specified by the command, the dPart field is used to indicate the offset within the *naturally aligned data block* specified by the address in the memory read command. For multiple responses to a single command, the CAPPTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TL may see the values of dPart returned in any order. When multiple responses are received for a memory read command, a combination of **mem_rd_response**, **mem_rd_response.ow**, and **mem_rd_fail** responses may be received. Taken together, all responses shall contain a combination of dLength and dPart to cover the command's dLength specification.

For the **pr_rd_mem** command and **config_read**, the dLength and dPart fields shall be specified as 64 bytes and offset at 0. A single response shall be returned. Violating this rule results in a *Bad response received* error event.

The Resp_code field indicates the type of failure being reported. The Resp_code field is specified in *Table 2-17*.

*Table 2-17. The Resp_code specification for **mem_rd_fail*** (Page 1 of 2)

Resp_code encode	Description
‘0000’ - ‘0001’	Reserved.
‘0010’	Retry request (rty_req). The read operation could not be serviced at this time. This is a long back off event. Use of this code point might be due to an event in the device that may require software intervention. The operation may be retried by the host.
‘0011’	Reserved.
‘0100’ - ‘0111’	Reserved
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

*Table 2-17. The Resp_code specification for **mem_rd_fail*** (Page 2 of 2)

Resp_code encode	Description
‘1000’	Data error (dError). The memory access completed. The data obtained by the AFU has been corrupted and is not correctable. Data is not sent to the host. This may be a recoverable error by retrying the operation. See the device documentation for additional information. **Engineering note** A dError condition may also be reported using **mem_rd_response**, **mem_rd_response.ow**, or **mem_rd_response.xw**, as appropriate for the TL command, and shall indicate that the data is bad using the bad data indication in the control flit as specified for the data carrier used.
‘1001’	Unsupported operand length. The operation specifies an operand length that is not supported by the device. A retry of the operation shall not be successful.
‘1010’	Reserved.
‘1011’	Bad address specification. The address specified is not naturally aligned on a boundary specified by the operand length. A retry of the operation shall not be successful.
‘1100’ - ‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be retried. This code point indicates that the state of the device due to the error occurrence does not allow a successful retry of the operation. This includes the following: • The device and function number specified in the address of a **config_read** is not recognized by the AFU. • **config_read** is issued with T=1. • Any other failure detected by the AFU that is not included in any of the specified response codes. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the device documentation.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

mem_rd_fail responds to the TL commands found in *Table 2-18*. For each command only the Resp_codes indicated with a Y may be used. Resp_code indicated with an N shall not be used.

*Table 2-18. **mem_rd_fail** Resp_code use by TL command*

TL command	rty_req (2)	dError (8)	Unsupported operand length (9)	Bad address specification (11)	Failed (14)
rd_mem	Y	Y	Y[1]	N	Y
pr_rd_mem	Y	Y	Y[1]	Y	Y
config_read	Y	Y	Y	Y	Y

[1] May occur during MMIO space read access only.

Memory read response	**mem_rd_response.ow**	‘0000 0011’
mem_response	TLX.vc.0, TLX.dcp.0	1

R	dP(2:0)			CAPPTag(15:0)																Opcode(7:0)							
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

In response to a memory read command initiated by the host, the AFU is returning data using 32-byte data carriers. The host can determine which command to associate the data with by using the CAPPTag provided with the command and returned with the response.

This response implies a data length of 32 bytes. Multiple response packets may be received for a single memory read command. The dPart (dP) indicates the offset within the *naturally aligned data block* specified by the address in the memory read command. For multiple responses to a single command, the CAPPTag is unchanged. That is, dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TL may see the values of dPart returned in any order. When multiple responses are received for a memory read command, a combination of **mem_rd_response**, **mem_rd_response.ow**, and **mem_rd_fail** responses may be received. Taken together, all responses shall contain a combination of dPart and implied lengths or dLength to cover the command's dLength specification.

For **pr_rd_mem**, the dPart field shall be specified as an offset at 0. A single response covers the entire operation.

This response is specified with immediate data. Credits for both the VC and DCP shall be obtained before this response is serviced by the TLX.

Memory write response	**mem_wr_response**	‘0000 0100’
mem_response	TLX.vc.0	1

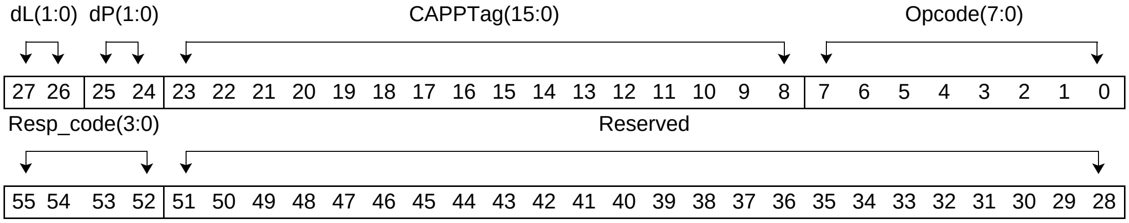
dL(1:0)		dP(1:0)		CAPPTag(15:0)																Opcode(7:0)							
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

This packet is used in response to a command that writes to memory. This response is used to indicate the successful completion of all or a portion of the operation. Data specified by this response is global visible. That is, a subsequent read shall see the new data.

For **write_mem** or **pad_mem**, the dLength (dL) and dPart (dP) fields specify how much of the write operation is being reported. A single response may cover the entire operation. For a single response, the dLength must match the dLength specified by the command, and dPart must indicate a starting offset of 0. Multiple response packets may be received for a single memory write command. When the dLength field in the response does not match the full amount of data specified by the command, the dPart field is used to indicate the offset from the starting address specified in the memory write command. For multiple responses to a single command, the CAPPTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TL may see the values of dPart returned in any order. When multiple responses are received for a memory write command, a combination of **mem_wr_response** and **mem_wr_fail** responses may be received. Taken together, all responses shall contain a combination of dLength and dPart to cover the command's dLength specification.

For the **pr_wr_mem**, **write_mem.be**, and **config_write** commands a single response shall be returned. The dLength and dPart fields shall be specified as 64 bytes and offset at 0.

Memory write failed	**mem_wr_fail**	‘0000 0101’
mem_response	TLX.vc.0	2



In response to a **write_mem** or **pad_mem** command initiated by the host, the AFU is indicating that the write failed. The host determines which command to associate with the failure by using the CAPPTag provided with the command and returned with the response.

The dLength (dL) and dPart (dP) fields specify how much of the write operation is being reported. A single response may cover the entire operation. For a single response, the dLength must match the dLength specified by the command, and dPart must indicate a starting offset of 0. Multiple response packets may be received for a single memory write command. When the dLength field in the response does not match the full amount of data specified by the command, the dPart field is used to indicate the offset from the starting address specified in the memory write command. For multiple responses to a single command, the CAPPTag is unchanged. That is, the dLength may vary and the dPart shall vary when multiple responses are returned for a single command. For multiple responses to a single command, there is no order requirement placed by the architecture. That is, the TL may see the values of dPart returned in any order. When multiple responses are received for a memory write command, a combination of **mem_wr_response** and **mem_wr_fail** responses may be received. Taken together, all responses shall contain a combination of dLength and dPart to cover the command's dLength specification.

For **config_write**, **pr_wr_mem** and **write_mem.be**, only one response shall be returned; dLength and dPart shall be specified as 64 bytes and offset at 0.

The Resp_code field indicates the type of failure being reported. The Resp_code field is specified in *Table 2-19*.

Table 2-19. The Resp_code specification for ***mem_wr_fail*** (Page 1 of 2)

Resp_code encode	Description
‘0000’ - ‘0001’	Reserved.
‘0010’	Retry request (rty_req). The write operation could not be serviced at this time. This is a long back off event. Use of this code point might be due to an event in the device that may require software intervention, or may be due to a hardware recovery mechanism. The operation may be retried by the host.
‘0011’	Reserved.
‘0100’ - ‘0111’	Reserved
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

Table 2-19. The Resp_code specification for ***mem_wr_fail*** (Page 2 of 2)

Resp_code encode	Description
‘1000’	Data error (dError). The AFU’s operation has completed.The data sent by the TL was corrupted prior to the completion of the operation.Changes, if any to the memory location specified by the response are globally visible. • Memory locations specified by the command’s PA and length that correspond to system memory space shall contain SUE data. • Memory locations specified by the command’s PA and length that correspond to either MMIO or configuration space may be unmodified, may contain undefined data, or may contain SUE data. The corruption of the data might have occurred anywhere in the AFU’s hardware or might have been detected by a bad data indication. **Engineering note** If an implementation is unable to modify the memory location specified by the command’s PA that correspond to system memory space to contain SUE data, the implementation shall not report a dError. The implementation shall report a Failed. **Engineering note** A dError condition may also be reported by the consumer of the data. That is, the reporting of the dError condition may be delayed until the data is consumed by a read operation. This requires that the actions taken when the error condition is detected either shall cause the memory location to contain SUE data or shall use an alternate method to report the data is invalid prior to or when it is consumed. When either of these methods are used, the AFU may response with **mem_wr_response**, instead of a **mem_wr_fail**.
‘1001’	Unsupported operand length. The operation specifies an operand length that is not supported by the device. A retry of the operation shall not be successful.
‘1010’	Reserved.
‘1011’	Bad address specification. The address specified is not naturally aligned on a boundary specified by the operand length. A retry of the operation shall not be successful.
‘1100 - ‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be retried.This code point indicates that the state of the device due to the error occurrence does not allow a successful retry of the operation. This includes the following: • The device and function number specified in the address of a **config_write** is not recognized by the AFU. • **config_write** specified with T=1. • A dError event was detected and the implementation is unable to modify the memory locations specified by the command’s PA and length that correspond to system memory space to contain SUE data. Changes, if any to the memory location specified by the response are globally visible. The memory location may be unmodified, or may contain undefined data. • Any other failure detected by the AFU that is not included in any of the specified response codes.The failure may cause the modification of the memory location specified by the command. Changes, if any to the memory location specified by the response are globally visible. The memory location may be unmodified, may contain undefined data, or may contain SUE data. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the device documentation.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

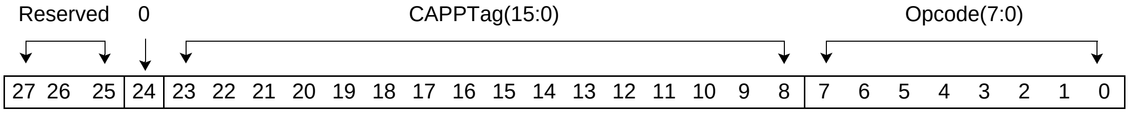
mem_wr_fail responds to the TL commands found in *Table 2-20*. For each command only the Resp_codes indicated with a Y may be used. Resp_code indicated with an N shall not be used.

*Table 2-20. **mem_wr_fail** Resp_code use by TL command*

TL command	rty_req (2)	dError (8)	Unsupported operand length (9)	Bad address specification (11)	Failed (14)
pad_mem	Y	N	Y[2]	Y	Y
write_mem	Y	Y	Y[2]	Y	Y
write_mem.be	Y	Y	N	Y	Y[2]
pr_wr_mem	Y	Y	Y[1]	Y	Y
config_write	Y	Y	Y	Y	Y

1. Unsupported operand length may occur only when target memory is defined as MMIO space and the command's specified pLength is not supported at the address specified.
2. May occur during MMIO address space write operation only.

Memory read response	**mem_rd_response.xw**	'0000 0111'
mem_response	TLX.vc.0, TLX.dcp.0	1

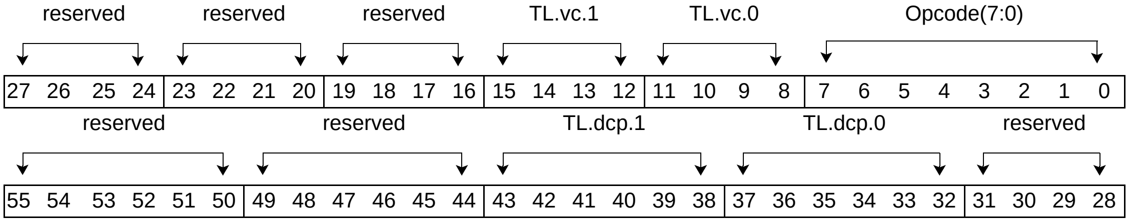


In response to a memory read command initiated by the host, the AFU is returning data using 8-byte data fields found in some control flits. The host can determine which command to associate the data with by using the CAPPTag provided with the command and returned with the response.

This response implies a data length of 8 bytes. A single response packet shall be received for a memory read operation when results are returned using this response packet. The full amount of data specified by the command is returned.

This response is specified with immediate data. Credits for both the VC and DCP shall be obtained before this response is serviced by the TLX.

Return TL credits	**return_tl_credits**	'0000 1000'
credit return	NA	2



This response packet is used by the TLX to return VC and DCP credits to the TL. There is no VC associated with this response, and credits are not required to service this response. Each TL.* field contains the number of credits being returned.

This response packet shall be placed only in slots 1 to 0 of any control flit using a template which specifies those slots as a 2-slot or larger location. The following exceptions apply:

- Control flits using template x’07’ may place a **return_tl_credits** response into slots 11 to 10.
- Control flits using template x’09’ may place a **return_tl_credits** response into slots 11 to 10.
- Control flits using template x’0B’ may place a **return_tl_credits** response into slots 13 to 12.

TL.vc.{0, 1} and TL.dcp.{0, 1} credits are returned. TL credits are for resources owned by the TLX that the TL consumes. The TLX controls the total number of credits for each of the VC and DCP it provisions the TL with.

Memory control operation done	**mem_cntl_done**	‘0000 1011’
message	TLX.vc.0	1

Resp_code	CAPPTag(15:0)	Opcode(7:0)
27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

In response to a **mem_cntl** command. The operation specified by the **mem_cntl** has completed as specified by the device manufacturer’s specification.

The Resp_code field specifies the type of error being reported. The Resp_code field is specified in *Table 2-21*.

*Table 2-21. The Resp_code specification for **mem_cntl_done***

Resp_code encode	Description
‘0000’	Complete. The operation has completed successfully.
‘0010’ - ‘1101’	Reserved
‘1100’ - ‘1101’	Reserved.
‘1110’	Failed. The operation has failed and cannot be retried.This code point indicates that the state of the device due to the error occurrence does not allow a successful retry of the operation. • The specification of the cmd_flag or object_handle fields in the **mem_cntl** command are invalid. • The operation has failed and cannot be retried for any reason. **Engineering Note** It is strongly recommended that an implementation provide error collection facilities to indicate the reason for the Resp_code = Failed. The specification of the error collection facility should be documented in the device documentation.
‘1111’	Reserved.
Note: The errors specified by Resp_code do not include the fatal error conditions described in *Table 7-1* on page 117.	

3. Virtual channel and data credit pool specification

Commands and responses are assigned to virtual channels (VCs) to allow ordering and the specification of servicing service queues and virtual queues. Credits to use these VCs are managed by the destination (where the resources are consumed) and are released to the source (where the command or response originates) when the resource is available. Each VC has its own credit pool, which may be varied by the destination. VCs in each direction are numbered; the value assigned is used only for differentiation between VC. Each VC credit permits the sending of one command or response.

The following VC descriptions use the specific command or response or the classification of the command or response. See the command and response descriptions in *Section 2 TL and TLX command and response specifications* on page 31 for command and response VC classifications.

VCs are specified between the TL and TLX and between the TLX and the TL. Ordering is maintained within a VC and is assured between these endpoints only. VCs cannot block each other; blocking within a VC may occur due to ordering requirements. With the exception of *Command ordering* on page 27, any queuing or ordering occurring in the upper protocol layers (host bus and AFU) is not assured to be retained. Synchronization points are managed at the interfaces between the TL and host bus protocol stack and between the TLX and AFU protocol stack.

Data credit pool (DCP) credits are required when a command or response has immediate data; that is, data that is associated with the sending of the command or response. For example, a write command has immediate data while a read command does not. Commands and responses with immediate data shall obtain the necessary credits for both the VC and DCP assigned in an atomic fashion.

For example, sending 128 bytes requires atomically obtaining two DCP credits when using 64-byte data flits. See *Section 5.1.3 Data transport, order, and alignment* on page 105 for details on the relationship between DCP credits and *data carrier* use. If the credits are not available, the command or response cannot be serviced. That is, the command or response shall not be placed into a DL frame for transmission. See the description of *DCP on page 18*.

The order of data sent to the destination is the same as the order of the data’s corresponding command or response that is sent to its destination. This allows the destination to use the arrival order of commands and responses with immediate data to associate the immediate data with its command or response.

Credits are released to the consumer of the credits by the owner of the resources that the credits represent. That is, TLX credits represent TL resources that are consumed by the TLX. TL credits represent TLX resources that are consumed by the TL. The responses used to return credits are **return_tlx_credits** and **return_tl_credits**. A compliant implementation shall:

- Provide a 16-bit counter to track credits available for use for each DCP and VC specified in the following sections.
- Provision a minimum of one and less than 64K credits for each VC specified in the following sections.
- Provision a minimum of four and less than 64K credits for each DCP specified in the following sections.

Developer Note

- The host is required to release credits only for the VC/DCP that will be used by the AFU. Releasing credits for VC/DCPs that are not going to be used might not be optimal because the released credits correspond to resources in the host that could have been used for actual command/data/response traffic from the AFU.
- The AFU is required to release credits only for the VC/DCP that will be used by the host. Releasing credits for VC/DCPs that are not going to be used might not be optimal because the released credits correspond to resources in the AFU that could have been used for actual command/data/response traffic from the host.

The credits required can be determined by knowing the AFU capabilities as described in *Section 1.2 Host operation modes* on page 26, the commands and responses used, and the associated VC and DCP.

Table 3-3 on page 95 specifies the assignment of VC and DCP to commands and responses.

Developer Note

The architecture permits an implementation to release a minimum-total of four DCP credits. Releasing a minimum-total of a single DCP credit was considered and discarded because this limits the data to a single 64-byte transfer. This was considered too restrictive and might not easily enable implementations to optimize for the data block size normally used by the implementation.

3.1 Virtual channel

Ordering shall be maintained between elements within a VC. Blocking between VCs shall not be permitted.

3.1.1 TLX command and response VC (TLX.vc)

The VC is directed from the AP to the host. Two VCs are specified {0,3}. VC credits are consumed by the TLX and are returned by the TL using **return_tlx_credits**.

Engineering Note

TLX VC credits represent resources in the TL used for processing TLX commands and responses. Each credit released by the TL represents a *unique* resource and shall not be shared with any other VC. Doing so would result in breaking the accounting rules implied by this specification. For example, if the TL used the same resource for two different VCs, the actual credit available for VCs using the shared resource would also be diminished, and the TLX would be unaware of this change.

The ability of the TL to return TLX VC credits shall not be dependent on any action taken by the TLX, including the return of TL credits.

3.1.2 TL command and response VC (TL.vc)

The VC is directed from the host to the AP. Two VCs are specified {0, 1}. VC credits are consumed by the TL and are returned by the TLX using **return_tl_credits**.

Engineering Note

TL VC credits represent resources in the TLX used for processing TL commands and responses. Each credit released by the TLX represents a *unique* resource and shall not be shared with any other VC. Doing so would result in breaking the accounting rules implied by this specification. For example, if the TLX used the same resource for two different VCs, the actual credit available for VCs using the shared resource would also be diminished, and the TL would be unaware of this change.

The ability of the TLX to return TL VC credits shall not be dependent on any action taken by the TL, including the return of TLX credits.

3.1.3 VC credit count specification

Table 3-1 specifies the maximum number of VC credits supported by an OpenCAPI-compliant device. A device is not required to release or support the maximum count. However, the consumer of the credit shall provide a counter that supports the architected maximum count.

Table 3-1. VC maximum credit count specification

VC	Maximum credit count
TLX.vc.0	64K-1
TLX.vc.3	64K-1
TL.vc.0	64K-1
TL.vc.1	64K-1

3.2 Data credit pool

3.2.1 TLX data DCP (TLX.dcp)

The data credit pool is used when moving immediate data from the AP to the host. Two DCPs are specified {0, 3}. DCP credits are consumed by the TLX and returned by the TL using **return_tlx_credits**.

Engineering Note

TLX DCP credits represent resources in the TL used for accepting data from the TLX. Each credit released by the TL represents a *unique* data resource and shall not be shared with any other DCP. Doing so would result in breaking the accounting rules implied by this specification. For example, if the TL used the same resource for two different DCPs, the actual credit available for DCPs using the shared resource would also be diminished, and the TLX would be unaware of this change.

See *Section 5.1.3 Data transport, order, and alignment* on page 105 for the association between a DCP credit and the immediate data associated with the command or response.

Resources for holding any meta-data associated with the data resource are accounted for by the DCP credit associated with the data itself.

The ability of the TL to return TLX DCP credits shall not be dependent on any action taken by the TLX, including the return of TL credits.

3.2.2 TL data DCP (TL.dcp)

This data credit pool is used when moving immediate data from the host to the AP. Two DCPs are specified {0, 1}. DCP credits are consumed by the TL and are returned by using the TLX using **return_tl_credits**.

Engineering Note

TL DCP credits represent resources in the TLX used for accepting data from the TL. Each credit released by the TLX represents a *unique* data resource and shall not be shared with any other DCP. Doing so would result in breaking the accounting rules implied by this specification. For example, if the TLX used the same resource for two different DCPs, the actual credit available for DCPs using the shared resource would also be diminished, and the TL would be unaware of this change.

See *Section 5.1.3 Data transport, order, and alignment* on page 105 for the association between a DCP credit and the immediate data associated with the command or response.

Resources for holding any meta-data associated with the data resource are accounted for by the DCP credit associated with the data itself.

The ability of the TLX to return TL DCP credits shall not be dependent on any action taken by the TL, including the return of TLX credits.

3.2.3 DCP credit count specification

Table 3-2 specifies the maximum number of DCP credits supported by an OpenCAPI-compliant device. A device is not required to release or support the maximum count. However, the consumer of the credit shall provide a counter that supports the architected maximum count.

Table 3-2. DCP maximum credit count specification

DCP	Maximum credit count
TLX.dcp.0	64K-1
TLX.dcp.3	64K-1
TL.dcp.0	64K-1
TL.dcp.1	64K-1

Table 3-3. Summary VC and DCP assignments (Page 1 of 2)

VC	Classification/ command	DCP	Comments	Command	Response
TL.vc.0	Touch response				X
TL.vc.0	DMA read (response)	TL.dcp.0	TL.dcp.0 is used only for **read_response** and is not used for **read_failed**.		X
TL.vc.0	Write response (OK and failed)				X
TL.vc.0	Interrupt and wake host thread responses		Message responses.		X
TL.vc.0	xlate_done		Asynchronous notification. Asynchronous command reporting the status of a previously AFU-initiated action (address translation touch).	X	
TL.vc.0	Interrupt ready		Asynchronous notification.	X	

Table 3-3. Summary VC and DCP assignments (Page 2 of 2)

VC	Classification/ command	DCP	Comments	Command	Response
TL.vc.1	MEM read both multiples of 64 bytes or partial commands			X	
TL.vc.1	MEM read		An MMIO read operation uses a **pr_rd_mem** CAPP command.	X	
TL.vc.1	MEM write	TL.dcp.1	An MMIO write operation uses a **pr_w_mem** CAPP command.	X	
TL.vc.1	Configuration register read			X	
TL.vc.1	Configuration register write	TL.dcp.1		X	
TLX.vc.0	MEM read response; for example, **mem_response**	TLX.dcp.0	**mem_rd_fail** does not use TLX.dcp.0.		X
TLX.vc.0	MEM write response				X
TLX.vc.3	Non-cacheable read and write operations - all forms			X	
TLX.vc.3	acTag management			X	
TLX.vc.3	Interrupts and wake host thread	TLX.dcp.3	Message commands. TLX.dcp.3 is used only for **intrp_req.d** commands.	X	
TLX.vc.3	**xlate_touch** - ATC prefetch			X	

3.3 TL Virtual channel and service queues

3.3.1 Host TLX command handling

Figure 3-1 on page 97 illustrates the steps a TLX command follows from the VC queue in the TL to its service queue. Commands are removed from the DL frame in slot order from a control flit and are loaded into the VC queue specified by the command. After the command reaches the head of the VC queue, it is examined.

1. Any error found in the command entry is noted. Errors found at this point in the flow shall not be reported until the command reaches the head of the service queue.
2. If the command is **assign_actag**, the command shall be removed from the VC queue and shall be serviced at this time. This service results in the update of the acTag table. Subsequent commands shall see the new state of the acTag table. The **assign_actag** command is removed from the head of the service queue.

   If the command is not **assign_actag**, the command shall be serviced as follows:

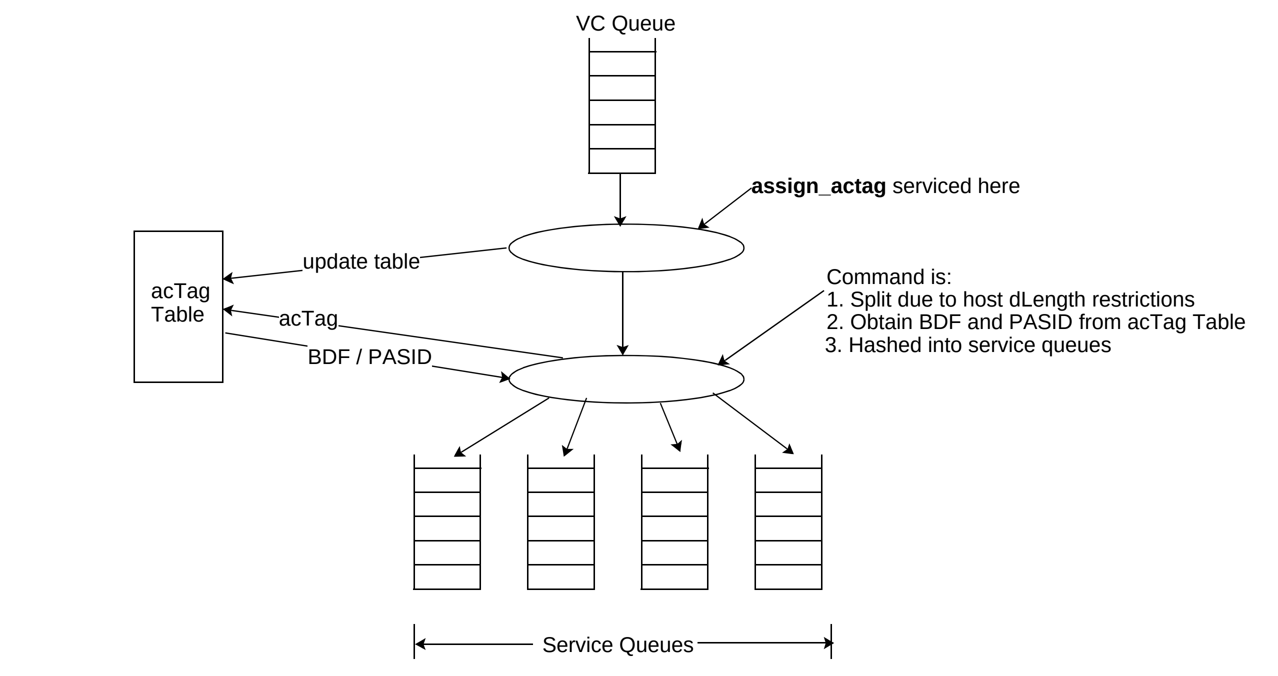
   a. Commands with a dLength specified that is larger than the host’s maximum supported length for a single operation are split into commands with dLength set equal to or less than the host’s maximum supported length. The address is adjusted for each command to account for the dLength serviced. The AFUTag, acTag, and stream_id are obtained from the original command. Split command entries are noted with the dPart value to be used when returning responses. Split commands are serviced based on increasing address order. All commands continue with the next step.

b. Using the acTag found in the command entry, the BDF and PASID are obtained. An error detected at this step is fatal. See the description of *Bad BDF and PASID combination on page 118* for additional details.

c. A VC- and implementation-specific hash is used to determine the service queue to add the command to. See the specification of the hash applied to the command in the definition of a *service queue on page 22*.

The command is then added to the service queue determined in step c and removed from the head of the VC queue.

Figure 3-1. *TL command flow from the VC queue to the service queue* TLX VC queues shown


VC Queue
assign_actag serviced here
update table
acTag Table
acTag
BDF / PASID
Command is:
1. Split due to host dLength restrictions
2. Obtain BDF and PASID from acTag Table
3. Hashed into service queues
Service Queues

As defined in *Terms* on page 17, the difference between a *virtual queue* and a *service queue* is that a *service queue* may contain multiple *virtual queues*.

The architectural model of a service queue specifies that the commands in the body of the service queue may receive the following services in the following order:

1. Error checking of the command.
2. Address translation as required by the command’s specification.

Errors occurring in either of these services shall be noted in the service queue entry and shall not be reported until the command reaches the head of the service queue. The results of the address translation shall be noted in the service queue entry.

The architectural model of a service queue specifies that the head of the service queue shall receive the following services in the following order:

1. Error checking of the command. The check may occur here, or the error noted in the entry may be used to determine the error checked state of the command.

2. Address translation as required by the command's specification. The translation may occur here, or may have previously occurred. The results of the translation noted in the entry may be used at this time.
3. When the previous two services are completed, the command shall be removed from the head of the service queue.

Step 3 requires that a command at the head of the service queue be processed with the indicated precedence as follows:

1. Failed due to an error, which may be due to either an error found with the command or failed address translation attempt.

   And

   – That error shall result in either a response being returned to the requester or an error event being asserted,

Or

2. The operation is completed[4] and a response shall be returned to the requester.

Or

3. Shall be dispatched to the host protocol layer.

Engineering Note

The architecture *requires* that the implementation of a service queue shall appear to behave as if the architectural model of a service queue is implemented.

3.3.2 Host TLX response handling

TLX responses are assigned to a VC and are removed from the DL frame in the same manner as TLX commands. Handling of TLX responses is simpler in that TLX.vc.0 which is used for TLX responses has no hash involved when selecting a service queue. Responses assigned to TLX.vc.0 are passed directly to a dedicated service queue and allowed to dispatch to the host interface.

3.4 Device TL virtual channel queues

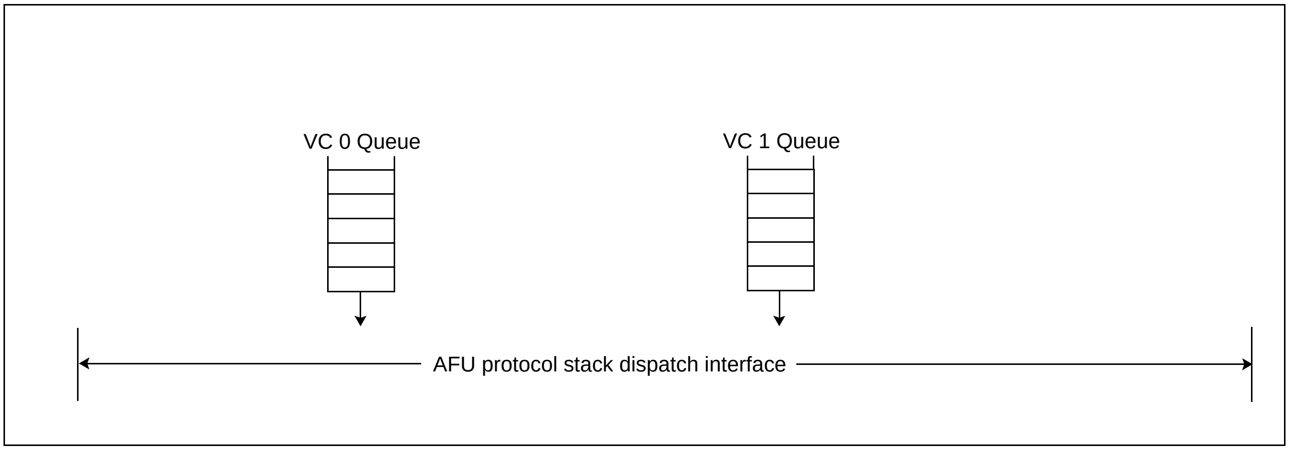
Figure 3-2 on page 99 shows the steps a TL VC queue entry follows from the time it is dequeued from the TL VC queue until it is dispatched to the AFU protocol stack interface. The TL VC queue entry can be a TL command- or response-packet. TL command- and response-packets are removed from the DLX frame in slot order from a control flit. They are loaded as queue entries into the TL VC queue specified by the command or response. After the queue entry reaches the head of the TL VC queue, it is examined.

1. Any error found in the queue entry shall be reported, and the entry shall be dequeued. If the TL VC queue entry is a TL command-packet, the errors shall be reported either by a response returned or by an error event. If the TL VC queue entry is a response-packet, errors shall be reported through error events such as *Bad response received on page 119*. Other error events are described in *Section 7.1* beginning on page 117.

[4] For example, the command required only an address translation action, such as **xlate_touch**, or the address translation was not successful or is required to be retried.

2. With error checking completed, the entry shall be dequeued and sent to the AFU protocol dispatch interface. At the dispatch interface, arbitration between the VC queues is implementation dependent and beyond the scope of this specification.

Figure 3-2. *TLX command and response flow from the VC to the AFU protocol stack* TL VC queues shown



3.5 Virtual channel dependency rules

Commands and responses are assigned to virtual channels. AFU and host designs are cautioned not to implement a design where there is a potential for a dead lock when forward progress of one command is dependent on the forward progress of another.

The TL specification specifies natural VC dependencies. For example a TLX **rd_wnitc** using TLX.vc.3 is dependent on a TL **read_response** using TL.vc.0 to complete the operation. Deadlock conditions can occur when:

- The host cannot respond to a TLX command without issuing a TL command and the host design does not allow the TL command to be issued.
- The AFU cannot respond to a TL command without issuing a TLX command and the AFU host design does not allow the TLX command to be issued.

Since the implementation of the host and the AFU are beyond the scope of this specification, the designers of the host and the AFU shall ensure that such dead lock conditions are prohibited by the implementation.

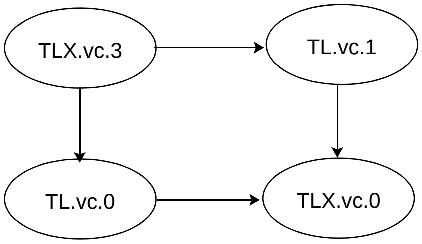
Figure 3-3 illustrates the interdependencies between VC that occur in the existing specification. An implementation, of either a host or device, shall not introduce dependencies not shown in *Figure 3-3*.

Graphically, the dependencies are shown as a line between two ovals. The ovals specify a virtual channel. For illustrative purposes, consider two ovals connected by an arrow. The tail of the arrow is connected to the oval indicating a TL- or TLX-packet using VC.a. The head of the arrow is connected to a TL- or TLX-packet using VC.b. Reading the graphic becomes:

> Servicing an incoming packet using VC.a requires issuing a packet using VC.b.

The intent of the architecture is to require implementations to insure that servicing an incoming packet using VC.a does not require the use of the same resources that are required to issue a packet using VC.b. This intent might restrict implementation choices.

Figure 3-3. *VC dependency graph*



4. The acTag table

The section provides the architectural specification of the acTag table. Implementations may choose to implement this table using any method. However, the externally observable behavior of the table and contents of the table shall, at a minimum, comply to this architectural specification.

The acTag table shall be included in the host's implementation and shall provide a minimum of one entry.

4.1 acTag table contents

Each entry of the acTag table (acTag entry) shall contain the following fields:

- 1-bit entry valid. Architected states are defined as {valid, invalid}
- 16-bit BDF
- 20-bit PASID

An implementation may choose to add additional fields to the acTag entry.

4.2 acTag table access

The OpenCAPI device maintains a copy of each acTag entry to determine the acTag value used in TLX commands specified with an acTag field.

The acTag table is accessed using the acTag as follows:

- Read access uses the acTag provided in a TLX command specified with an acTag field as an index into the table to locate the acTag entry to be read. Reading an acTag entry returns the entry valid indication and, when valid, a BDF and PASID.
- Write access uses the acTag provided in the **assign_actag** command as an index into the table to locate the acTag entry to be updated. The command provides a BDF and PASID, which are loaded into the acTag entry. Successful completion of the write access sets the entry valid bit to the valid state.

4.2.1 Error cases when accessing the acTag table

Error	Action
acTag entry not valid	This is a fatal error. See "acTag specified in the command points to an invalid entry" in *Table 7-1* on page 117.
Address context not valid	This is determined either at the time the acTag entry is created or when the acTag entry is used to obtain the address context. This is a fatal error. See "Bad BDF and PASID combination" in *Table 7-1* on page 117.

4.3 acTag entry management

The entries of the acTag table are managed by the attached OpenCAPI device. The OpenCAPI device shall maintain its own mapping between an acTag and the contents of the acTag entry held in the host's acTag table.

During configuration, the host writes to the OpenCAPI device's configuration space to indicate the maximum size of the acTag, which indirectly specifies the size of the host's acTag table. An acTag size of 0 indicates a single entry acTag table, and the only valid acTag value is '0'. The OpenCAPI device may use any value within the allowed range to specify an acTag entry and subsequently refer to that entry using a TLX command specifying the acTag.

The OpenCAPI device learns its bus number from the address specified in a **config_write**, T=0 command. Device and function numbers are assigned by the OpenCAPI device and discovered by the host during configuration. See the specification of **config_write** for the format of the address field that contains the bus, device and function numbers.

The OpenCAPI device is configured with one or more PASIDs during initialization and operation.

After an acTag entry is set to a valid state, it is set to an invalid state only by the host upon detection of a link failure that requires resetting the OpenCAPI interface and the OpenCAPI device.

Engineering note

In a host implementation, a configuration register would be useful to vary the size of the acTag table, which allows stress testing of the design. A proposed specification of the configuration register follows:

The register contains a value N where the size of the acTag table is 2^N and the acTag range is limited, as described previously, to a range of $0..2^{N-1}$.

Permitted access methods for this configuration register, as well as the register contents, are specified by the platform architecture.

5. DL frame format

The TL and TLX contain framer and parser functions that work with the DL frame format. The DL frame format is specified as a set of 64-byte flits. There are two types of flits:

- Control flits. The control flit contains TL command/response content and DL content. The DL content contains several DL-generated subfields including the CRC that covers the control flit and any *preceding* data flits. There are fields in the DL content that are generated by the TL. For more information, see *Section 5.1.1 DL content* on page 104.
- Data flits. There are 0 to 8 data flits between each control flit.

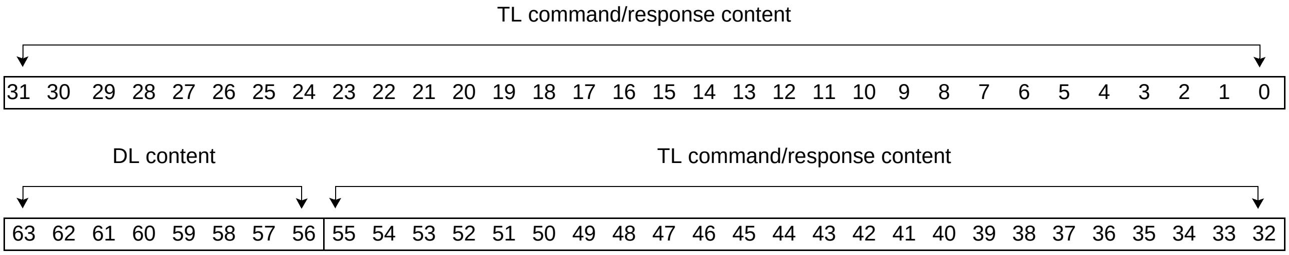
Table 5-1. DL frame format showing CRC and “bad data flit” coverage

← Bytes(63:0) →	
DL content	TL command/response/32-, 8-byte data content
Data flit 0	
Data flit 1	
Data flit 2	
Data flit 3	
Data flit 4	
Data flit 5	
Data flit 6	
Data flit 7	
DL content	TL command/response/32-, 8-byte data content
Data flit 0	
Data flit 1	
DL content	TL command/response/32-, 8-byte data content
DL content	TL command/response/32-, 8-byte data content

Table 5-1 uses color to illustrate the coverage of the CRC found in the DL content of the control flit. The CRC covers the control flit that it is contained in and all *previous*, if any, data flits. The DL content found in the control flit also contains “bad data flit indicators” for the previous data flits. The 32- and 8-byte data fields carry bad data indicators for the associated data field in the control flit. A control flit specified by the TL shall always follow data flits as shown in *Table 5-1*. The DL may insert DL-idle-control flits when the TL interface to the DL is idle.

The transmit order in *Table 5-1* is from right to left and top to bottom. That is, data flits are transmitted in increasing address order. Control flits follow this convention.

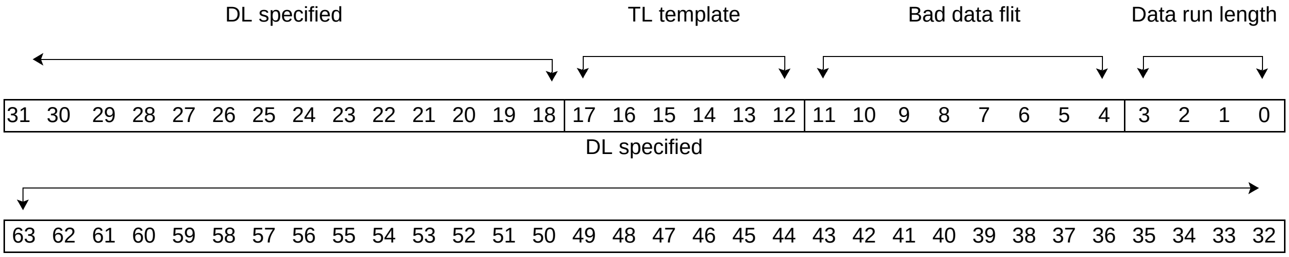
5.1 DL frame control flit (64 bytes)



Bytes	Field name	Description
55:0	TL command/response content	This field contains information provided by the TL. This 448-bit field (447:0) is comprised of sixteen 28-bit slots. One or more slots comprise either a null entry, TL command packet, TL response packet, metadata, extended metadata, or data with the location and length specified by the TL template. See *Section 5.1.2 TL command/response content* on page 105 for slot layout information.
63:56	DL content	This field contains information added by both the TL and the DL layer. See *Section 5.1.1 DL content* for the specification of this field.

5.1.1 DL content

This field is bytes 63:56 of the DL frame control flit.

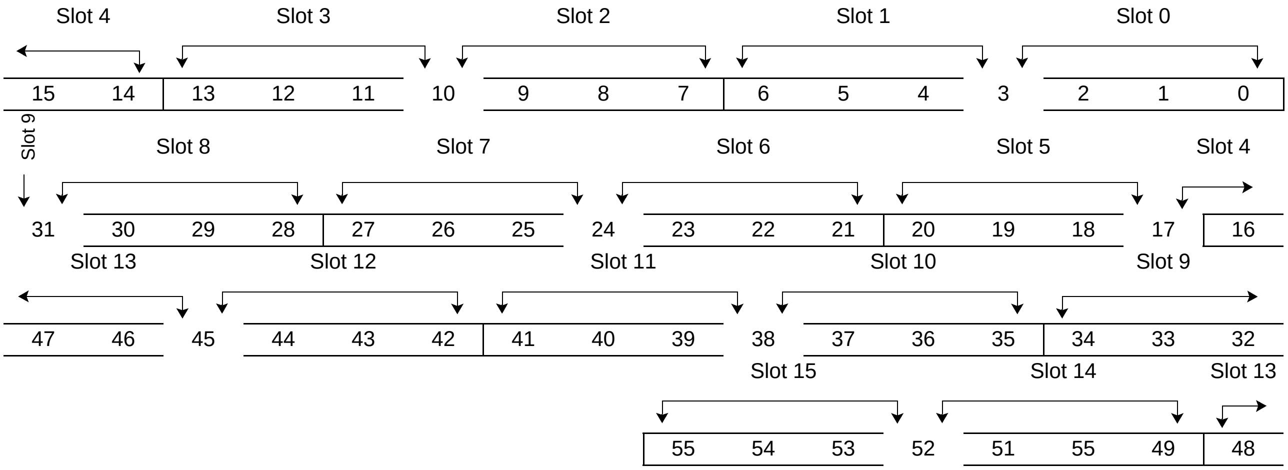


Bit	Field name	Description (Page 1 of 2)
3:0	Data run length	This 4-bit field indicates the number of data flits until the next control flit. A value of 0 indicates that the next flit is a control flit. Valid values are {0...8}.
11:4	Bad data flit indication	This 8-bit field indicates that data flits received prior to this control flit contain bad data and shall not be used without being marked as bad (for example, mark as SUE). Each bit corresponds to one data flit (for example, bit 0 corresponds to data flit 0, which is the first data flit following the previous control flit). See *Table 5-1* on page 103 to match up the data flit being reported to the bit in this field. 11 Data flit 7 is in error. 10 Data flit 6 is in error. 9 Data flit 5 is in error. 8 Data flit 4 is in error. 7 Data flit 3 is in error. 6 Data flit 2 is in error. 5 Data flit 1 is in error. 4 Data flit 0 is in error.
17:12	TL template	This 6-bit field specifies the locations of opcodes found in the 448-bit TL command and response content field. See*Section 6 TL and TLX template specifications* on page 108 for the specification of this field.

Bit	Field name	Description (Page 2 of 2)
63:18	DL specified	The specification for this 46-bit field is found in the OpenCAPI DL specification. This field is expected to contain the CRC, ACK, and ACK count information.

5.1.2 TL command/response content

Slots are packed into the DL control flit as shown in the following figure. Each slot occupies 3.5 bytes (28 bits). The slot number and control flit byte are shown. The number of slots used by a command or response can be found in the specification of the command or response.



5.1.3 Data transport, order, and alignment

Data is transported between the TL and TLX using *data carriers,* which are specified as 64-byte data flits, or 32- or 8-byte data fields in control flits. To transport data, one or more DCP credits shall be obtained atomically when obtaining the VC credit required to send a command or response. A DCP credit is associated with at most 64 bytes of data and may be associated with 32- or 8-bytes of data.

1. When a single command or response specifies 64-bytes of immediate data, a single DCP credit is required. Either a 64-byte data flit or 2 32-byte data carriers may be used.
2. When a single command or response specifies 128- or 256-bytes of immediate data, 2 and 4 DCP credits are required. Multiple 64-byte data flits, or multiple 32-byte data carriers, or a combination of 64- and 32-byte data carriers may be used.
3. When a single command or response specifies 32 or fewer bytes of immediate data, a single DCP credit is required and a single data carrier shall be used. This applies to commands and responses with a pLength field specified, dot-be commands, 8-byte data carrier use, and dot-ow responses.

A data carrier shall be associated with a single command or response. Multiple data carriers may be associated with a single command or response.

Within each control flit, there is an order to the commands and responses as shown in *Section 5.1.2 TL command/response content* on page 105. Commands and responses loaded into lower numbered slots are ordered before commands and responses loaded into higher numbered slots. Data shall be loaded into data carriers by the TL in the same order as the commands and responses are specified in the control flits. Data carried by control flits that are loaded into lower numbered slots are ordered before data carried by higher numbered slots. Data carried in a control slot is ordered before the data carried in data flits that follow. In control flits with data carriers, the commands or responses are treated as if they precede the data. For example, a command or response that specifies immediate data may find the data in the same control flit as the command or response.

For each command and response associated with data, there is an implied length or a dLength field and a dPart field specified. These are used to pull the data out of the data carriers and associate the data with the command or response. Each data carrier is examined using the length information and the data is associated with a single command or response. (Additional association can be made using the AFU or CAPP tag provided to locate the machine associated with the command or response.) In some cases (for example, **dma_pr_w**), the dLength field is implicit and is specified as a single data carrier. The minimum size of the data carrier is determined by the command's pLength field.

Data shall be address aligned within a 64-, 32- and 8-byte data carrier. That is, this architecture treats a data carrier as if it were a memory-mapped naturally aligned data block, where each byte of data is loaded into the data carrier based on the address of the byte being loaded. When a command or response with immediate data uses multiple data carriers, the data shall be loaded in increasing address order. That is, offset 0 from the address specified by the command or response, and adjusted by the dPart field, shall be loaded first and the remaining data is loaded in increasing address order.

When the data is not associated with an address, the data shall be placed starting at byte 0 of the data carrier and increasing byte locations until all the data has been loaded into the data carrier. A command or response with this type of data shall use only a single data carrier. Additional restrictions might be found in the command and response description.

Engineering note

There are naturally occurring cases when all bytes of a data carrier are not fully specified by the command or response associated with the data. For example, the data associated with a **dma_pr_w** does not specify all bytes within a 64-byte data flit. That is, there are bytes within the data carrier that are defined by the write operation based on pLength and starting address, and there are bytes that are undefined by the architecture.

It is strongly recommended that the architecturally undefined data bytes found within a data carrier do not contain information associated with any application other than the application associated with the command or response and is limited to the permissions granted to the application.

The method used to ensure that the contents of undefined data locations within a data carrier are not from a different process is determined by the implementation. Suggested methods include, but are not limited to the following:

- Set all undefined byte locations to zero.
- Set all undefined byte locations to a fixed or random non-zero pattern of bits
- Replicate the content of defined byte locations to undefined byte locations.

The architecture does not provide architectural conformance statements regarding the contents of undefined byte locations within a data carrier other than the conformance statement specified by the definition of the architectural term "*undefined*".

A 64-byte data flit may be used with any type of data that is specified for one of the following:

- Any command
- Responses not specified as dot-ow and dot-xw

  One to four data flits may be used to provide data associated with an address. The number of data flits is dependent on the number of bytes specified by the command or response associated with the immediate data. When multiple data flits are used, the data flits shall be loaded in increasing address order based on the dLength and dPart specified in the command or response.

A 32-byte data field found in a control flit shall be used to carry data associated with an address only. One or more 32-byte data fields may be associated with a single command or response. When multiple 32-byte data carriers are used, data shall be loaded in increasing address order based on the dLength and dPart specified in the command or response.

- dot-ow responses are associated with a single 32-byte data field.
- dot-xw responses shall not use 32-byte data fields as a data carrier.

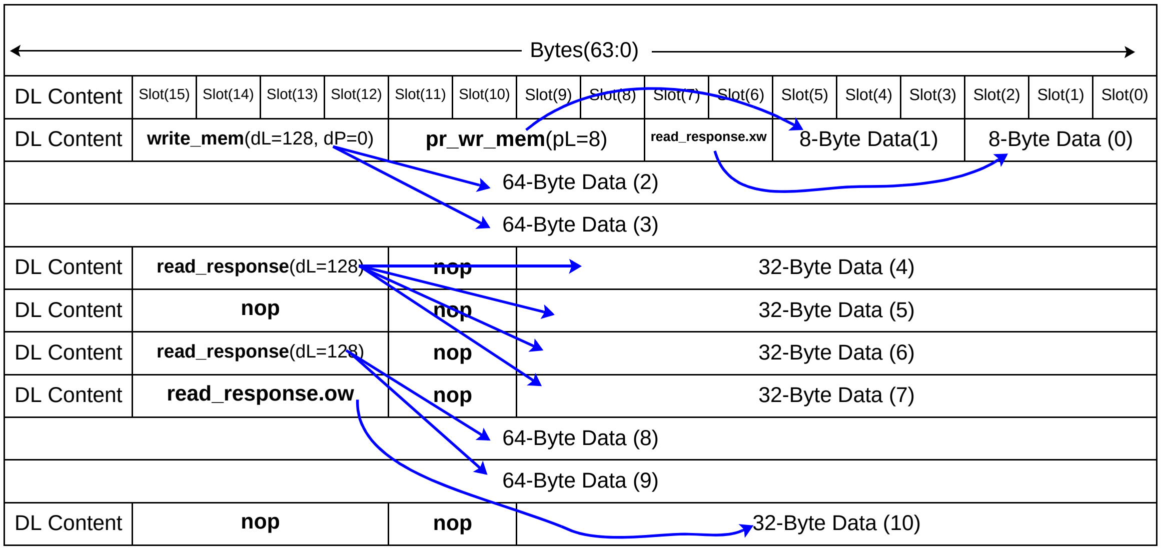
A combination of 64- and 32-byte data carriers may be used for commands and responses with 64- 128- or 256-bytes of immediate data specified. 64-byte carriers shall be loaded on aligned 64 byte boundaries. 32-byte data carriers shall be loaded on aligned 32-byte boundaries. Note that this restricts when an implementation is allowed to switch between using 64- and 32-byte data carriers since the architecture requires that the data be loaded in increasing address order regardless of the mix of data carriers used.

An 8-byte data field found in a control flit may be used to carry data associated with an address only. Only one 8-byte data field shall be associated with a command or response.

- dot-xw responses shall use only 8-byte data fields as a data carrier.

Table 5-2 illustrates how a command and response stream's data might be packed into a series of control and data flits following the ordering rules for data.

Table 5-2. DL frame loading to illustrate data ordering



Bytes(63:0)																
DL Content	Slot(15)	Slot(14)	Slot(13)	Slot(12)	Slot(11)	Slot(10)	Slot(9)	Slot(8)	Slot(7)	Slot(6)	Slot(5)	Slot(4)	Slot(3)	Slot(2)	Slot(1)	Slot(0)
DL Content	**write_mem**(dL=128, dP=0)				**pr_wr_mem**(pL=8)				**read_response.xw**		8-Byte Data(1)			8-Byte Data (0)		
64-Byte Data (2)																
64-Byte Data (3)																
DL Content	**read_response**(dL=128)				**nop**		32-Byte Data (4)									
DL Content	**nop**				**nop**		32-Byte Data (5)									
DL Content	**read_response**(dL=128)				**nop**		32-Byte Data (6)									
DL Content	**read_response.ow**				**nop**		32-Byte Data (7)									
64-Byte Data (8)																
64-Byte Data (9)																
DL Content	**nop**				**nop**		32-Byte Data (10)									

6. TL and TLX template specifications

This specification defines the allowed placement formats of TL/TLX packets, metadata, extended metadata, 8- and 32-byte data fields within the DL/DLX frame's control flit. The allowed formats are captured in four capability descriptions defined in *Table 6-1* on page 109.

A TL template field is specified within the DL content of a DL packet's control flit. The DL content of the control flit is shown in *Section 5.1 DL frame control flit (64 bytes)* on page 104. The DLX frame has the same format as the DL frame.

The TL architecture specifies all template capability descriptions and specifies a number for each specification that is used in the TL template field. When transmitting a packet:

- The TL shall place the TL transmit template number used to form the control flit of the DL frame.
- The TLX shall place the TLX transmit template number used to form the control flit of the DLX frame.

The template capabilities specify the legal locations of one or more TL/TLX command or response packets' starting slot[5] as well as the contiguous number of slots used by the packet. In addition to TL/TLX command and response packets, the templates specify the legal locations of metadata and data and the contiguous number of slots used by the metadata and data fields. The format of the data fields is also specified. Unused control flit slots are reserved. That is, unused control flit slots shall be set to an all zero state when transmitted and shall not be examined on receipt for any purpose other than CRC checking.

The template restricts the maximum length of TL/TLX command or response packets placed in the control flit. The template specification permits a smaller packet to be placed into a larger specification footprint. For example, a six-slot template specification may be filled with a 4-, 2-, or 1-slot packet. The state of the unused slots is undefined.

Developer Note

Allowing smaller TL packets to occupy larger packet-specified locations helps to reduce the number of template specifications.

A template may further restrict the TL/TLX command or response packet placed into packet locations. These restrictions include and are not limited to the following:

- Command
- Response
- VC used
- DCP requirement (present or absent)

Table 6-1 on page 109 defines the template capabilities.

[5] Bits 27:0 of the TL/TLX packet specification.

Table 6-1. Template capability definitions

Capability	Definition	Specified by
TLX receive template	Specifies the templates that the OpenCAPI device supports when receiving DL frames.	OpenCAPI device
TL transmit template	Specifies the templates that the host supports when transmitting a DL frame to the OpenCAPI device.	Host
TL receive template	Specifies the templates that the host supports when receiving DLX frames.	Host
TLX transmit template	Specifies the templates that the OpenCAPI device supports when transmitting a DLX frame to the host.	OpenCAPI device

The intersection of the TL transmit template capability and the TLX receive template capability shall not be a null set. All OpenCAPI devices shall support TLX receive template x‘00’.

The intersection of the TLX transmit template capability and the TL receive template capability shall not be a null set. All OpenCAPI hosts shall support TL receive template x‘00’.

See the host’s platform architecture for additional information about how the receive and transmit capabilities are resolved and what is stored into the OpenCAPI device’s configuration space.

Table 6-2 defines the terms used in *Section 6.1* and *Section 6.2*.

Table 6-2. Terms used in template capability specifications (Page 1 of 2)

Term	Width (bits)	Description
<n>-slot <type> packet	n*28	The number <n> of slots specified for a packet <type> of either a TL or TLX.
Data(x:y)		Indicates a data field of x + 1 - y bits in length.
xmeta	72	Extended-metadata. A 72-bit field associated with a 32 byte *naturally aligned data block*. Extended metadata is placed in the control flit carrying the 32-byte data block. The specification of the extended-metadata is outside the scope of this architecture and is found in the host and OpenCAPI device’s documentation.
Meta	7	Metadata. A 7-bit field associated with a *naturally aligned data block*. The size of the data block is implementation dependent. Metadata may be associated with blocks smaller than the implementation-specified size during data transfer. The specification of the metadata is outside the scope of this architecture and is found in the host and OpenCAPI device’s documentation. The implementation defines the transformation of the metadata associated with a smaller block when merged into a larger data block. For example, the implementation’s data block size might be 64 bytes and an update to an 8-byte *naturally aligned data block* might be required. When the 8-byte block is provided with metadata, the implementation determines how the metadata provided with the 8-byte block is used to transform the metadata associated with the 64-byte block. • mdf(n) indicates metadata associated with the n^{th} data flit following the control flit containing the metadata. Each field is 7 bits wide. • meta(6:0) specifies a single metadata field associated with data found in the control flit.
R		Indicates a reserved bit in a slot. • R(x:y) specifies x + 1 - y reserved bits • R(0) specifies a single reserved bit

Table 6-2. Terms used in template capability specifications (Page 2 of 2)

Term	Width (bits)	Description
V	2	A 2-bit field indicating that the associated data is valid and if it is bad. Bit Description 0 Bad data indication. This bit is valid only when bit 1 is set to ‘1’. 0 The associated data is good. 1 The associated data is bad. 1 Valid field indication. 0 The associated data is not valid. 1 The associated data and bit 0 are valid.

6.1 TLX receive and TL transmit template capability specification

Table 6-3. TLX receive/TL transmit template (Page 1 of 3)

Slot #	**x‘00’**	x‘01’	x‘02’	x‘03’
0	**return_tlx_credits**[a]	4-slot TL packet	2-slot TL packet	4-slot TL packet
1				
2	reserved		2-slot TL packet	
3				
4	6-slot TL packet	4-slot TL packet	2-slot TL packet	6-slot TL packet
5				
6			2-slot TL packet	
7				
8		4-slot TL packet	2-slot TL packet	
9				
10	reserved		2-slot TL packet	6-slot TL packet
11				
12		4-slot TL packet	2-slot TL packet	
13				
14			2-slot TL packet	
15				

Table 6-3. TLX receive/TL transmit template (Page 2 of 3)

Slot #	x‘04’	x‘05’	x‘06’	x‘07’[b]
0	2-slot TL packet	2-slot TL packet	2-slot TL packet	Data(27:0)
1				Data(55:28)
2	mdf(3) || mdf(2) || mdf(1) || mdf(0)	mdf(3) || mdf(2) || mdf(1) || mdf(0)	mdf(3) || mdf(2) || mdf(1) || mdf(0)	Data(83:56)
3	mdf(7) || mdf(6) || mdf(5) || mdf(4)	mdf(7) || mdf(6) || mdf(5) || mdf(4)	mdf(7) || mdf(6) || mdf(5) || mdf(4)	Data(111:84)
4	4-slot TL packet	1-slot TL packet	6-slot TL packet	Data(139:112)
5		1-slot TL packet		Data(167:140)
6		1-slot TL packet		Data(195:168)
7		1-slot TL packet		Data(223:196)
8	4-slot TL packet	1-slot TL packet		Data(251:224)
9		1-slot TL packet		mdf(1) || mdf(0) || R(0) || V(1:0) || meta(6:0) || Data(255:252)
10		1-slot TL packet	6-slot TL packet	2-slot TL packet
11		1-slot TL packet		
12	4-slot TL packet	4-slot TL packet		4-slot TL packet
13				
14				
15				

Table 6-3. TLX receive/TL transmit template (Page 3 of 3)

Slot #	x‘08’[c]	x‘09’[d]	x‘0A’[e]	x’0B’[f]
0	Data(27:0)	Data(27:0)	Data(27:0)	Data(27:0)
1	Data(55:28)	Data(55:28)	Data(55:28)	Data(55:28)
2	R(10:0) || V(1:0) || meta(6:0) || Data(63:56)	Data(83:56)	Data(83:56)	Data(83:56)
3	Data(27:0)	Data(111:84)	Data(111:84)	Data(111:84)
4	Data(55:28)	Data(139:112)	Data(139:112)	Data(139:112)
5	R(10:0) || V(1:0) || meta(6:0) || Data(63:56)	Data(167:140)	Data(167:140)	Data(167:140)
6	2-slot TL packet	Data(195:168)	Data(195:168)	Data(195:168)
7		Data(223:196)	Data(223:196)	Data(223:196)
8	4-slot TL packet	Data(251:224)	Data(251:224)	Data(251:224)
9		mdf(1) || mdf(0) || R(0) || V(1:0) || meta(6:0) || Data(255:252)	xmeta(23:0) || Data(255:252)	xmeta(23:0) || Data(255:252)
10		2-slot TL packet	xmeta(51:24)	xmeta(51:24)
11			R(5:0) || V(1:0) || xmeta(71:52)	R(5:0) || V(1:0) || xmeta(71:52)
12	4-slot TL packet	1-slot TL packet	4-slot TL packet	2-slot TL packet
13		1-slot TL packet		
14		1-slot TL packet		1-slot TL packet
15		1-slot TL packet		1-slot TL packet

[a] Template x’0’ slots 0 and 1 shall contain either a **nop** or **return_tlx_credits**

[b] Template x’07’ is limited to a two data flit run length. Slot 9 contains the metadata for data flits 0 and 1 as shown.

[c] Template x‘08’ should only be used when there is at least one 8-byte data carrier valid. That is, a single 8-byte data shall be placed in slots 0 to 2. Violating this rule may cause a *Bad template usage* error to be reported. Metadata fields are not provided in this template. This template shall only be used when metadata is not required for the data in the data flits following the control flit using this template.

[d] Template x’09’ is limited to a two data flit run length. Slot 9 contains the metadata for data flits 0 and 1 as shown.

[e] Template x’0A’ specifies extended metadata associated with the 32-byte data carrier in the control flit. This template shall only be used when metadata is not required for the data in the data flits following the control flit using this template.

[f] Template x’0B’ specifies extended metadata associated with the 32-byte data carrier in the control flit. This template shall only be used when metadata is not required for the data in the data flits following the control flit using this template.

6.2 TL receive and TLX transmit template capability specification

Table 6-4. TL receive/TLX transmit template (Page 1 of 2)

Slot #	**x‘00’**	x‘01’	x‘02’	x‘03’
0	**return_tl_credits**[a]	4-slot TLX packet	2-slot TLX packet	4-slot TLX packet
1				
2	reserved		2-slot TLX packet	
3				
4	6-slot TLX packet	4-slot TLX packet	2-slot TLX packet	6-slot TLX packet
5				
6			2-slot TLX packet	
7				
8		4-slot TLX packet	2-slot TLX packet	
9				
10	reserved		2-slot TLX packet	6-slot TLX packet
11				
12		4-slot TLX packet	2-slot TLX packet	
13				
14			2-slot TLX packet	
15				

Slot #	x‘04’	x‘05’	x‘06’	x‘07’[b]
0	2-slot TLX packet	2-slot TLX packet	2-slot TLX packet	Data(27:0)
1				Data(55:28)
2	mdf(3) || mdf(2) || mdf(1) || mdf(0)	mdf(3) || mdf(2) || mdf(1) || mdf(0)	mdf(3) || mdf(2) || mdf(1) || mdf(0)	Data(83:56)
3	mdf(7) || mdf(6) || mdf(5) || mdf(4)	mdf(7) || mdf(6) || mdf(5) || mdf(4)	mdf(7) || mdf(6) || mdf(5) || mdf(4)	Data(111:84)
4	4-slot TLX packet	1-slot TLX packet	6-slot TLX packet	Data(139:112)
5		1-slot TLX packet		Data(167:140)
6		1-slot TLX packet		Data(195:168)
7		1-slot TLX packet		Data(223:196)
8	4-slot TLX packet	1-slot TLX packet		Data(251:224)
9		1-slot TLX packet		mdf(1) || mdf(0) || R(0) || V(1:0) || meta(6:0) || Data(255:252)
10		1-slot TLX packet	6-slot TLX packet	2-slot TLX packet
11		1-slot TLX packet		
12	4-slot TLX packet	4-slot TLX packet		4-slot TLX packet
13				
14				
15				

Table 6-4. TL receive/TLX transmit template (Page 2 of 2)

Slot #	x‘08’[c]	x‘09’[d]	x‘0A’[e]	x’0B’[f]
0	Data(27:0)	Data(27:0)	Data(27:0)	Data(27:0)
1	Data(55:28)	Data(55:28)	Data(55:28)	Data(55:28)
2	R(10:0) || V(1:0) || meta(6:0) || Data(63:56)	Data(83:56)	Data(83:56)	Data(83:56)
3	Data(27:0)	Data(111:84)	Data(111:84)	Data(111:84)
4	Data(55:28)	Data(139:112)	Data(139:112)	Data(139:112)
5	R(10:0) || V(1:0) || meta(6:0) || Data(63:56)	Data(167:140)	Data(167:140)	Data(167:140)
6	2-slot TLX packet	Data(195:168)	Data(195:168)	Data(195:168)
7		Data(223:196)	Data(223:196)	Data(223:196)
8	4-slot TLX packet	Data(251:224)	Data(251:224)	Data(251:224)
9		mdf(1) || mdf(0) || R(0) || V(1:0) || meta(6:0) || Data(255:252)	xmeta(23:0) || Data(255:252)	xmeta(23:0) || Data(255:252)
10		2-slot TLX packet	xmeta(51:24)	xmeta(51:24)
11			R(5:0) || V(1:0) || xmeta(71:52)	R(5:0) || V(1:0) || xmeta(71:52)
12	4-slot TLX packet	1-slot TLX packet	4-slot TLX packet	2-slot TLX packet
13		1-slot TLX packet		
14		1-slot TLX packet		1-slot TLX packet
15		1-slot TLX packet		1-slot TLX packet

[a] Template x’00’ slots 0 and 1 shall contain either a **nop** or **return_tl_credits**.

[b] Template x’07’ is limited to a two data flit run length. Slot 9 contains the metadata for data flits 0 and 1 as shown.

[c] Template x‘08’ should only be used when there is at least one 8-byte data valid. A single 8-byte data shall be placed in slots 0 to 2. Violating this rule may cause a *Bad template usage* error to be reported. Metadata fields are not provided in this template. This template shall only be used when metadata is not required for the data in the data flits following the control flit using this template.

[d] Template x’09’ is limited to a two data flit run length. Slot 9 contains the metadata for data flits 0 and 1 as shown.

[e] Template x’0A’ specifies extended metadata associated with the 32-byte data carrier in the control flit. This template shall only be used when meta data is not required for the data in the data flits following the control flit using this template.

[f] Template x’0B’ specifies extended metadata associated with the 32-byte data carrier in the control flit. This template shall only be used when meta data is not required for the data in the data flits following the control flit using this template.

6.3 Control-flit rate capability

Each receive and transmit template capability specification has an associated control-flit rate capability. When a template is used to format a control flit, the template’s associated control flit rate capability specifies when the next control flit may be sent.

The control-flit rate capability controls the DL flit spacing between control flits. The configuration space for this capability provides 4 bits.

- A value of x‘0’ indicates that a control flit may be sent in the following *flit-cycle*.
- A value of x‘1’ indicates that the cycle following the control flit shall contain either a null control flit or a data flit.

In general, a value of “n” indicates that there shall be a gap of “n” 64-byte flits before the next control flit can be sent. During the gap, either null control flits or data flits shall be inserted.

A null control flit is defined as using template x‘00’. The 6-slot packet contains a 1-slot null command, and the remaining five slots are undefined. A return credit response found in slots 0 and 1 may be used to return credits.

Engineering note

At the start of initialization of an OpenCAPI device, the flit rate capability is unknown since the configuration space has not yet been examined. Template x’00’ is used to issue **config_read** commands to determine the device’s capabilities.

Until the device’s capabilities are determined, template 0 shall be used and the control flit rate capability shall be assumed to be x’F’.

6.4 Metadata capability

Templates x‘04’ through x‘06’ allow the specification of metadata associated with up to eight data flits. Templates x‘07’ and ‘09’allow specification of metadata associated with up to two data flits. To enable this, a metadata capability is specified in the configuration space for the host and the OpenCAPI device.

Templates x’0A’ and x‘0B’ allows the specification of extended-metadata for the 32-byte data carrier specified in the control flit. To enable this, an extended-metadata capability is specified in the configuration of the host and the configuration space for the OpenCAPI device.

7. Error detection

This section identifies errors and classes of errors detected by the TL and TLX. Error notifications and the collection of error signatures are specified.

- The host or device may provide:
  - Additional error signature information for error events defined by this architecture
  - Additional error events that are implementation-specific.

Specification of these implementation-specific error signature and error event extensions might be found in either the host’s platform architecture, the host’s user’s guide, or the manufacturer’s documentation provided with the OpenCAPI device.

Implementation-specific fatal error events shall be summarized in the *AFU Fatal error detected* and the *Host Fatal error detected* error events specified by this specification. This specification provides these summary error events to expose the existence of fatal implementation-specific error events and are included in conformance tests. Note that conformance testing may not test the underlying implementation-specific error events, but shall ensure that the architecturally specified error events are tested.

Actions taken by hypervisors, operating systems, firmware, or device drivers when error notifications are asserted are beyond the scope of this architecture.

Error events are specified in *Table 9-1* on page 213 using the following format:

Error event name	Description of error event • Action taken	
	Error signature:	The minimum set of information captured by a compliant design. **Engineering Note** The error signatures specified in *Table 9-1* on page 213 shall be accessible and may be obtained for diagnostic use by an examination of hardware facilities and may require additional host- or device-specific software manipulation.
	Error Class	Specifies the class and architecture conformance requirements.

Error classes are specified as follows:

- *Correctable error events* are error conditions that the hardware can recover without any loss of function, state, or data.
- *Fatal error events* are error conditions that result in the unrecoverable loss of function, state, or data. Continued use of the link and attached device might not be safe. A reset might be required to return the link and device to a safe operational state. The architecture does not place conformance requirements on an implementation once a fatal error event has occurred. That is, continued use of the link may occur and the results of operations after a fatal error are undefined.

  The TL shall report the detection of a fatal error to the device using the method described by the OpenCAPI DL specification.

  The TLX shall report the detection of a fatal error to the host using the method described by the OpenCAPI DL specification.
- *Non-fatal error events* are error conditions that affect the operation of a single transaction. The link is considered to be operationally safe. The results of the transaction might not be as intended. That is, the

results of the operation are undefined. Devices associated with the error might require a reset. Devices not associated with the error are not affected by the error. Continued use of the link may occur and the results of operations after a non-fatal error shall conform to the requirements of the architecture.

Error events are assigned error types and may be assigned an error subtype. These assignments are found in bold text in the description of the error event.

Error events are assigned an error class and a conformance requirement.

- *Required error events* are demanded by the architecture and shall be included in any architecture conformance testing. All error events specified as required shall be included in both the TL and TLX implementation unless otherwise specified.
- *Optional error events* are not required by the architecture. Careful reading of the description of the error events assigned as optional is strongly recommended since detection may be required due to architecturally required actions to be taken. Conformance testing may indicate the presence or absence of the hardware's capability to detect and report the error event.

7.1 Error events

The following error events are specified

acTag specified in a command is outside the configured specification set	acTag specified in the command points to an invalid entry	AFU Fatal error detected
Age out specified for **xlate_touch.n**	Bad BDF and PASID combination	Bad data flit indication error
Bad data received	Bad opcode and template combination	Bad response received
Bad template x‘00’ format	Bad template usage	Control flit overrun
Host Fatal error detected		
Illegal return credit command location	log2_page_size specification in **xlate_touch** is bad.	Missing Metadata
PA specified is out of bounds	Reserved field not transmitted as 0	Reserved field value used
Reserved opcode used	Returned credit overflows credit counter	Unexpected data carrier
Unsupported template format		

Table 7-1. Error event specification (Page 1 of 7)

<table>
<tr><th>Error event</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3">acTag specified in a command is outside the configured specification set</td><td colspan="2">On receipt of a TLX command packet containing an acTag field, it is determined that the acTag is specified outside the configured specification set.
• The operation is aborted without changes to the machine state, and a <b>malformed packet error type 2 event</b> is asserted.</td></tr>
<tr><td><b>Error signature:</b></td><td>opcode(7:0), AFUTag(15:0)
<b>Engineering note</b>
When the TLX command is <b>assign_actag</b>, the AFUTag in the error signature is reserved.</td></tr>
<tr><td><b>Error Class:</b></td><td>Fatal/Required (TL only)</td></tr>
</table>

Table 7-1. Error event specification (Page 2 of 7)

<table>
<tr><th>Error event</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3">acTag specified in the command points to an invalid entry</td><td colspan="2">On receipt of a TLX command packet containing an acTag field, the entry in the acTag table is examined and found to be marked invalid.
The operation is aborted without changes to the machine state, and an **address context error type 0 event** is asserted.</td></tr>
<tr><td>**Error signature:**</td><td>opcode(7:0), AFUTag(15:0), acTag(11:0)</td></tr>
<tr><td>**Error Class:**</td><td>Fatal/Required (TL only)</td></tr>
<tr><td rowspan="3">AFU Fatal error detected</td><td colspan="2">An AFU and device implementation-specific fatal error.
This error is specified by the device manufacturer.</td></tr>
<tr><td>**Error signature:**</td><td>See the manufacturer’s device documentation.</td></tr>
<tr><td>**Error Class:**</td><td>Fatal/Optional (TLX only)</td></tr>
<tr><td rowspan="3">Age out specified for **xlate_touch.n**</td><td colspan="2">An **xlate_touch.n** is specified with a command flag of “age out”. This is a nonsensical combination. The dot-n directive is ignored, and the operation proceeds to completion.
• An **xlate_touch error type 1 event** may be asserted.</td></tr>
<tr><td>**Error signature:**</td><td>AFUTag(15:0)</td></tr>
<tr><td>**Error Class:**</td><td>Non-fatal/Optional (TL only)</td></tr>
<tr><td rowspan="3">Bad BDF and PASID combination</td><td colspan="2">On receipt of a TLX command packet containing an acTag field, the entry in the acTag table is found to be valid. The BDF and PASID specified are not valid for use.
The operation is aborted without changes to the machine state, and an **address context error type 1 event** is asserted.</td></tr>
<tr><td>**Error signature:**</td><td>opcode(7:0), AFUTag(15:0), acTag(11:0)</td></tr>
<tr><td>**Error Class:**</td><td>Fatal/Required (TL only)</td></tr>
<tr><td rowspan="3">Bad data flit indication error</td><td colspan="2">On the receipt of a control flit, the bad data flit field indicates that a data flit is bad and is located beyond the scope of the control flit. That is, the control flit indicates that n data flits follow, and the bad data flit field indicates that data flit n or above is in error. In the following valid combinations, an ‘x’ indicates that the field may take on either a 0 or 1 state.
<table>
<tr><th>Data run length</th><th>Bad data flit</th></tr>
<tr><td>x‘0’</td><td>‘0000 0000’</td></tr>
<tr><td>x‘1’</td><td>‘0000 000x’</td></tr>
<tr><td>x‘2’</td><td>‘0000 00xx’</td></tr>
<tr><td>x‘3’</td><td>‘0000 0xxx’</td></tr>
<tr><td>x‘4’</td><td>‘0000 xxxx’</td></tr>
<tr><td>x‘5’</td><td>‘000x xxxx’</td></tr>
<tr><td>x‘6’</td><td>‘00xx xxxx’</td></tr>
<tr><td>x‘7’</td><td>‘0xxx xxxx’</td></tr>
<tr><td>x‘8’</td><td>‘xxxx xxxx’</td></tr>
</table>
• A **malformed control flit error type 3** event is asserted.
The bad data flit information beyond the scope of the control flit is ignored.</td></tr>
<tr><td>**Error signature:**</td><td>The bad data flit and data run length fields are captured. These are found in the DL content of the control flit found in bits 11:0 as shown in *Section 5.1.1 DL content* on page 104.</td></tr>
<tr><td>**Error Class:**</td><td>Non-fatal/Optional (TL and TLX)</td></tr>
</table>

Table 7-1. Error event specification (Page 3 of 7)

Error event	Description	
Bad data received	On processing data flits, a data flit is marked bad by the bad data flit indication field found in the control flit as described in *Section 5.1.1 DL content* on page 164. • A **bad data flit error event** may be asserted. The command or response associated with the bad data is provided with the data and the bad data indication. The bad data indication shall be propagated to the final destination of the data. It is strongly recommended that the error propagation be implemented in a fashion that allows for error isolation; that is, first error incidence reporting.	
	Error signature:	Data is associated with: TLX response packet: opcode(7:0), CAPPTag(15:0). TLX command packet: opcode(7:0), acTag(11:0), AFUTag(15:0). TL response packet: opcode(7:0), AFUTag(15:0). TL command packet: opcode(7:0), CAPPTag(15:0)
	Error Class:	Non-fatal/Optional (TL and TLX)
Bad opcode and template combination	The format of the control flit specified by the template is in error. Opcodes specified indicate packet sizes larger than allowed by the template found in the control flit. This error is detected in both the TL and TLX. All commands or responses identified in the control flit are aborted and do not cause any machine state changes, and a **malformed control flit error type 2 event** is asserted.	
	Error signature:	• Template (5:0) found in the control flit. • The opcode (7:0) found in the control flit where it was determined that the template packet size rules were violated. • The slot location, a 4-bit field, where it was determined that the template packet size rules were violated.
	Error Class:	Fatal/Required. (TL and TLX)
Bad response received	**TL:** On receipt of a TLX response packet, the CAPPTag is first examined to determine if a prior command has been issued using the CAPPTag found in the response packet. It is reported as a bad response received variant 0 if the CAPPTag has not been used. If the response packet is not a variant 0, the response opcode is checked to determine if it is a valid response for the command opcode used. It is reported as a bad response received variant 1 if it is not. **TLX:** On receipt of a TL response packet, the AFUTag is first examined to determine if a prior command has been issued using the AFUTag found in the response packet. It is reported as a bad response receive variant 0 if the AFUTag has not been used. If the response packet is not a variant 0, the response opcode is checked to determine if it is a valid response for the command opcode used. It is reported as a bad response received variant 1 if it is not. • The actions specified by the completion of the command due to a correct response are aborted, and a **malformed packet error type 5 event** is asserted. The state machine representing the command source is left in an undefined state. The undefined state of the machine is bounded, that is, the state is known to the implementation and the actions taken by the state machine in this architecturally undefined state is predictable by the implementation.	
	Error signature:	TL: CAPPTag(15:0), variant(0). For a variant 1, the command opcode(7:0) is also provided. TLX: AFUTag(15:0), variant(0). For a variant 1, the command opcode(7:0) is also provided. In the error signature, the variant(0) field is set to the variant error type. • variant(0) = ‘0’ when the error is variant 0. • variant(0) = ‘1’ when the error is variant 1.
	Error Class:	Fatal/Required (TL and TLX)

Table 7-1. Error event specification (Page 4 of 7)

Error event	Description	
Bad template x‘00’ format	The format of the control flit, specified as using the x‘00’ template, does not match the template x‘00’ format. Slot 0 does not contain either a **nop**, or **return_tlx_credits** (detected by the TLX), or **return_tl_credits** (detected by the TL), opcode. • Any commands or responses found in the control flit are aborted and do not cause any machine state changes. A **malformed control flit error type 0 event** is asserted.	
	Error signature:	Slot 0 (27:0) contents
	Error Class:	Fatal/Required (TL and TLX)
Bad template usage	Template x‘08’ is improperly used. 1. The template contains two 8-byte data fields, and the field starting in slot 0 is unused (the valid bit is set to 0). (TL) A **malformed control flit error type 3 event** is asserted. Any valid data is used. Command and response packets, if any, are not dropped. 2. The data is associated with a fetch and swap operation (**amo_rw**, cmd_flag = {x‘8’...x‘A’}) (TL) A **malformed control flit error type 4 event** is asserted. Any valid data associated with the control flit is dropped.	
	Error signature:	Template (5:0) found in the control flit.
	Error Class:	1. Non-fatal/Optional 2. Fatal/Required
Control flit overrun	The destination is unable to accept a subsequent control flit. A possible cause is a violation of the control flit rate capability for the prior control flit's template. • The incoming control flit is discarded. The machine state is unchanged. A **control flit overrun error** event is asserted.	
	Error signature:	None
	Error Class:	Fatal/Required (TL and TLX)
Host Fatal error detected	A TL and host implementation-specific fatal error. This error is specified by the host manufacturer.	
	Error signature:	See the manufacturer’s device documentation.
	Error Class:	Fatal/Optional (TL only)
Illegal return credit command location	**TL**: **return_tl_credits** shall be found only in the following slots based on the template used: x‘07’ slots (11:10) x‘09’ slots (11:10) x’0B slots(13:12) All other templates slots(1:0) **TLX**: Regardless of the template used, **return_tlx_credits** shall be found only in slots 1:0. • The credit return, as specified by the command, may occur. A **malformed packet error type 3 event** shall be asserted.	
	Error signature:	Template (5:0) found in the control flit; slot where return credit opcode was found (3:0).
	Error Class:	Fatal/Required (TL and TLX)

Table 7-1. Error event specification (Page 5 of 7)

Error event	Description	
log$_2$_page_size specification in **xlate_touch** is bad.	This error is detected when an **xlate_touch** is specified with a cmd_flag specification of age-out, and the page size specified by the ATC entry found does not match the page size specified by the command’s log$_2$_page_size field. • See *Figure 2-1 Address translation sequence: **xlate_touch*** on page 111 for actions taken when there is a mismatch. • An **xlate_touch error type 0 event** may be asserted.	
	Error signature:	acTag(11:0)
	Error Class:	Non-fatal/Optional (TL only)
Missing Metadata	The host and attached OpenCAPI device have been configured to use metadata, and metadata has not been provided for a command or response that is specified with immediate data. The assertion of this error excludes **config_write** and **config_read** operations. These operations tolerate the presence of metadata. That is, the use of metadata for **config_write** and **config_read** operations is not defined by this architecture and an error shall not be reported. When missing metadata is determined, the following actions may be used to allow the link to continue operation. TL: The host may • provide non-destructive metadata to the data block. The value used is beyond the scope of the TL architecture. • mark the data as bad. TLX: The OpenCAPI device may • provide non-destructive metadata to the data block. The value used is beyond the scope of the TL architecture. • mark the data as bad. A **Missing metadata error** is asserted.	
	Error signature:	TL: AFUTag; TLX: CAPPTag.
	Error Class:	Non-fatal/Optional (TL and TLX)
PA specified is out of bounds	A command specifies a PA that is determined to be out of bounds for the AFU$_M$. For TL commands, the host has specified a PA that is outside the AFU$_{M1}$ PA range. • The operation is aborted without changes to the machine state. A **PA specification error event** is asserted.	
	Error signature:	opcode(7:0), PA(63:0)
	Error Class:	Fatal/Required (TLX only)
Reserved field not transmitted as 0	On the receipt of a command or response packet, it is determined that a field specified by the architecture as reserved does not contain 0. • The packet is used.That is, the operation specified by the command or response occurs normally. This is an architecture conformance violation. • A **malformed packet error type 4 event** is asserted.	
	Error signature:	None.
	Error Class:	Non-fatal/Optional (TL and TLX)

Table 7-1. Error event specification (Page 6 of 7)

<table>
<tr><th>Error event</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3">Reserved field value used</td><td colspan="2">On the receipt of a command or response packet, it is determined that a field specification contains an architecturally reserved value. If multiple fields contain reserved values, only one field is reported.
• The operation is aborted and the machine state is unchanged. A <b>malformed packet error type 1 event</b> is asserted.
The following fields have reserved values. Detection occurs at the receiver of the command or response packet.:
• <i>cmd_flag</i>. Detected by TL and TLX.
• <i>dLength</i>. Detected by TL and TLX.
• <i>dPart(1:0)</i> or <i>dPart(2:0)</i> detected by TL and TLX.
• <i>pLength</i>. Detected by TL and TLX.
• <i>Resp_code</i>. Detected by TL and TLX.</td></tr>
<tr><td><b>Error signature:</b></td><td>opcode(7:0), starting (LSb) field offset within the packet. For example, the acTag in a <b>rd_wnitc</b> TLX command packet has a field offset of 24.</td></tr>
<tr><td><b>Error Class</b></td><td>Fatal/Required (TL and TLX)</td></tr>
<tr><td rowspan="3">Reserved opcode used</td><td colspan="2">On the receipt of a command or response packet, the opcode field is examined and found to be a value reserved by the architecture.
• The packet is dropped, the machine state is unchanged. A <b>malformed packet error type 0 event</b> is asserted.</td></tr>
<tr><td><b>Error signature:</b></td><td>opcode(7:0)</td></tr>
<tr><td><b>Error Class:</b></td><td>Fatal/Required (TL and TLX)</td></tr>
<tr><td rowspan="3">Returned credit overflows credit counter</td><td colspan="2">On processing of a <b>return_tl_credits</b> or <b>return_tlx_credits</b> response packet, it is determined that the addition of the credits specified by the response will cause the counter to overflow.
• The counter may increment and is allowed to saturate. That is, the counter shall not wrap. A <b>credit return error event</b> is asserted.</td></tr>
<tr><td><b>Error signature:</b></td><td>opcode(7:0), specification of the counter or counters associated with the error using the following format:
<table>
<tr><td>TL:</td><td>R</td><td>R</td><td>dcp.1</td><td>dcp.0</td><td>R</td><td>R</td><td>R</td><td>vc.1</td><td>vc.0</td></tr>
<tr><td>TLX:</td><td>dcp.3</td><td>R</td><td>R</td><td>dcp.0</td><td>R</td><td>vc.3</td><td>R</td><td>R</td><td>vc.0</td></tr>
<tr><td></td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td></tr>
<tr><td></td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table></td></tr>
<tr><td><b>Error Class:</b></td><td>Fatal/Required (TL and TLX)</td></tr>
</table>

Table 7-1. Error event specification (Page 7 of 7)

Error event	Description	
Unexpected data carrier	The destination has detected that the accumulated number of data carriers has exceeded the amount of data expected. The expected data count is determined by an examination of the commands and responses received that specify immediate data. The ordering requirements specifying commands or responses with immediate data are received before the data is found in *Section 5.1.3 Data transport, order, and alignment* on page 105. **TL:** TLX packets received by the TL with immediate data specify the amount of data expected from the TLX. TLX packets that specify immediate data: **mem_rd_response** **dma_w** **dma_w.n** **dma_w.be** **dma_w.be.n** **dma_pr_w** **dma_pr_w.n** **amo_rw** **amo_rw.n** **amo_w** **amo_w.n** **intrp_req.d** **mem_rd_response.ow** **mem_rd_response.xw** **TLX:** TL packets received by the TLX with immediate data specify the amount of data expected by the TL. TL packets that specify immediate data: **read_response** **write_mem** **write_mem.be** **pr_wr_mem** **config_write** **read_response.ow** **read_response.xw** • The unexpected data carrier's contents may be discarded. An **unexpected data carrier error** event is asserted. **Developer Note** Expected data can be counted by determining the number of DCP credits required to send the data as specified by the command or response. Counting DCP credits can be split out by data credit pool number or aggregated. Counts are incremented as expected-data-information, as described above, is observed. Counts are decremented as the data is received. That is, the DCP count of expected data is decremented as the data carriers arrive. When the count drops below zero, an unexpected data carrier error is detected. Other implementation specific methods may also be used.	
	Error signature:	None
	Error Class:	Fatal/Required (TL and TLX)
Unsupported template format	An unsupported template is specified in a received control flit. • Any commands or responses identified in the control flit are aborted. Changes to the machine state are undefined. A **malformed control flit error type 1 event** is asserted.	
	Error signature:	Template (5:0) found in the control flit.
	Error Class:	Fatal/Required (TL and TLX)

8. OpenCAPI profiles

A device shall use an OpenCAPI profile to specify groups of commands, responses, and templates supported by the device. Each row in the following tables identifies an architectural feature, and each column indicates the content of a profile. The full specification of a profile is comprised of the same column from all tables in this section.

Within each profile a feature is marked using the notation found in *Table 8-1*. The specification of the compliance notation is taken from the view of the consumer of the command or response packet. That is, a command that is specified as mandatory requires that the consumer of the command shall process and execute the command per the architecture specification. Since the architecture specifies any responses or errors that are reported, those features become mandatory as well.

Table 8-1. Feature compliance requirement notation

Support requirement notation	Description
	(blank / empty) No conformance requirement, no recommendation guidance provided.
M	Mandatory
M.cx	Mandatory when the AFU is C1. Otherwise it is unsupported (U).
M.ir	Mandatory when the AFU issues any form of **intrp_req**. Otherwise it is unsupported (U).
M.mx	Mandatory when the AFU is M1. Otherwise it is unsupported (U).
M.tp	Mandatory when the required template is supported by the TL and TLX. Otherwise it is unsupported (U).
M.wht	Mandatory when the TLX issues **wake_host_thread**. Otherwise it is unsupported (U).
M.xt	Mandatory when **xlate_touch** is issued by the TLX. Otherwise it is unsupported (U).
O	Optional. This feature may be supported. Conformance evaluation to determine the presence of the feature and when present shall test its architectural compliance.
O.E	Compliance specification for endianness support. Either big or little endian data formats used in atomic* class commands may be supported. An implementation shall support one of the formats.
U	Unsupported. This feature is not included in conformance evaluation. Use of a command or response noted as unsupported may result in a fatal error event.

Table 8-2 specifies the compliance requirements for the TLX and AFU accepting and processing a command from the TL and host.

Table 8-2. Profile specifications for TL commands

TL command	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
config_read	M	M
config_write	M	M
intrp_rdy	M.ir	M.ir
mem_cntl	U	O
nop	M	M
pad_mem	U	O
pr_rd_mem	M,mx	M
pr_wr_mem	M.mx	M
rd_mem	M.mx	M
rd_pf	U	M
write_mem	M.mx	M
write_mem.be	M.mx	M
xlate_done	M.cx	U

Table 8-3 specifies the compliance requirements for the TL and host accepting and processing a command from the TLX and AFU.

Table 8-3. Profile specifications for TLX commands (Page 1 of 2)

TLX command	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
amo_rd	M	U
amo_rd.n	M	U
amo_rw	M	U
amo_rw.n	M	U
amo_w	M	U
amo_w.n	M	U
assign_actag	M	M
dma_pr_w	M	U
dma_pr_w.n	M	U
dma_w	M	U
dma_w.be	M	U
dma_w.be.n	M	U
dma_w.n	M	U
intrp_req	M	M
intrp_req.d	M	M
nop	M	M

Table 8-3. Profile specifications for TLX commands (Page 2 of 2)

TLX command	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
pr_rd_wnitc	M	U
pr_rd_wnitc.n	M	U
rd_wnitc	M	U
rd_wnitc.n	M	U
wake_host_thread	M	U
xlate_touch	M	U
xlate_touch.n	M	U

Table 8-4 specifies the compliance requirements for the TLX and AFU accepting and processing a response from the TL and host. Note that in most cases, TL responses are due to TLX commands issued to the host.

Table 8-4. Profile specifications for TL responses

TL response	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
intrp_resp	M.ir	M.ir
nop	M	M
read_failed	M.cx	U
read_response	M.cx	U
read_response.ow[a]	U	U
read_response.xw[b]	U	U
return_tlx_credits	M	M
touch_resp	M.xt	U
wake_host_resp	M.wht	U
write_failed	M.cx	U
write_response	M.cx	U

[a] Use of **read_response.ow** is dependent on support of either templates x'07' or x'09'.

[b] Use of **read_response.xw** is dependent on support of template x'08'.

Table 8-5 specifies the compliance requirements for the TL and the host accepting and processing a response from the TLX and AFU. Note that in most cases, TLX responses are due to TL commands issued to the TLX.

Table 8-5. Profile specifications for TLX responses (Page 1 of 2)

TLX response	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
mem_cntl_done	U	O
mem_rd_fail	M	M
mem_rd_response	M	M

Table 8-5. Profile specifications for TLX responses (Page 2 of 2)

TLX response	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
mem_rd_response.ow[a]	U	M.tp
mem_rd_response.xw[b]	U	M.tp
mem_wr_fail	M	M
mem_wr_response	M	M
nop	M	M
return_tl_credits	M	M

[a] Use of **mem_rd_response.ow** is dependent on support of either templates x’07’ or x’09’.

[b] Use of **mem_rd_response.xw** is dependent on support of template x’08’.

Table 8-6 and *Table 8-7* add template capability specifications to profiles. As discussed in *Section 6 TL and TLX template specifications* on page 108, the host’s platform architecture provides additional information about how receive and transmit capabilities are resolved between the host and the attached OpenCAPI device. Other than the mandatory support for transmitting and receiving template x‘00’ template control flits, the profile specifications for templates provides guidance as to the recommended templates an implementation should support and is not a conformance requirement. The templates marked as Optional are recommended. See *Section 6.1 TLX receive and TL transmit template capability specification* on page 110 and *Section 6.2 TL receive and TLX transmit template capability specification* on page 113. All host and devices shall support template x‘00’.

Table 8-6 specifies the requirements and recommendations for the

- TLX to accept a control flit using the specified template.
- TL to transmit a control flit using the specified template.

Table 8-6. Profile specifications for TLX receive/TL transmit templates (Page 1 of 2)

TLX receive/TL transmit template	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
x‘00’	M	M
x‘01’	O	O
x‘02’	O	
x‘03’	O	
x‘04	U	O
x‘05’	U	
x‘06’	U	
x‘07’[a]	U	O
x‘08’[b]	U	
Template notes: To support metadata, an implementation must support one of the following templates: x’04, x’05’, or x’06.		

Table 8-6. Profile specifications for TLX receive/TL transmit templates (Page 2 of 2)

TLX receive/TL transmit template	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
x’09’[c]	U	
x‘0A’[d]	U	O
x‘0B’[e]	U	
Template notes: To support metadata, an implementation must support one of the following templates: x’04, x’05’, or x’06.		

[a] Specifies a 32-byte data carrier. This template is used to support dot-ow response forms.

[b] Specifies two 8-byte data carriers. This template is used to support dot-xw response forms.

[c] Specifies a 32-byte data carrier. This template is used to support dot-ow response forms.

[d] Specifies a 32-byte data carrier with extended-metadata.

[e] Specifies a 32-byte data carrier with extended-metadata.

Table 8-7 specifies the requirements and recommendations for the

- TL to accept a control flit using the specified template.
- TLX to transmit a control flit using the specified template.

Table 8-7. Profile specifications for TL receive/TLX transmit templates

TL receive/TLX transmit template	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
x‘00’	M	M
x‘01’	O	O
x‘02’	O	
x‘03’	O	
x‘04	U	
x‘05’	U	O
x‘06’	U	
x‘07’[a]	U	
x‘08’[b]	U	
x’09’[c]	U	O
x‘0A’[d]	U	
x‘0B’[e]	U	O

[a] Specifies a 32-byte data carrier. This template is used to support dot-ow response forms.

[b] Specifies two 8-byte data carriers. This template is used to support dot-xw response forms.

[c] Specifies a 32-byte data carrier. This template is used to support dot-ow response forms.

[d] Specifies a 32-byte data carrier with extended-metadata.

[e] Specifies a 32-byte data carrier with extended-metadata.

Table 8-8 specifies the operation modes recommended to be supported by the host and the OpenCAPI device for different interface classifications and is not a conformance requirement. The operation modes marked as Optional are recommended. The definition of the host operation modes are found in *Section 1.2 Host operation modes* on page 26.

Table 8-8. Profile specifications host operation modes

AFU type	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
AFU_{C0}	O	
AFU_{C1}	O	O
AFU_{M0}	O	
AFU_{M1}	O	M

Table 8-9 adds page size support specification to profiles. The profile specification for page size provides guidance as to the recommended page sizes supported by the host's ATC and is not a conformance requirement. Address translation and ATC are discussed in *Section 1.6* on page 30. Page sizes marked as Optional are recommended.

Table 8-9. Profile specifications supported page size

page size	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
4K	O	
64K	O	

Table 8-10 specifies the compliance requirements for the TLX and AFU accepting and processing commands and responses from the TL and host with the dLength specified.

Table 8-10. Profile specifications supported dLength by TLX

dLength specification	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
64	M	M
128	M	M
256	O	O

Table 8-11 specifies the compliance requirements for the TL and host accepting and processing commands and responses from the TLX and AFU with dLength specified.

Table 8-11. Profile specifications supported dLength by TL

dLength specification	Device interface class, OpenCAPI 3.0	OMI, OpenCAPI 3.1
64	M	M
128	M	M
256	M	M

Table 8-12 specifies the compliance requirements for the TL and host accepting and processing atomic.* class commands based on the endianness of the data. The *E* field in the command specifies the endianness of the data. See *Table 8-3* for the compliance requirements for the TL and host accepting and processing these commands.

Table 8-12. Profile specifications support of endianness data format by the TL

TLX atomic* class command	Device interface class, OpenCAPI 3.0		OMI, OpenCAPI 3.1	
	E=0	E=1	E=0	E=1
amo_rd	O.E	O.E	U	U
amo_rd.n	O.E	O.E	U	U
amo_rw	O.E	O.E	U	U
amo_rw.n	O.E	O.E	U	U

Appendix A. AP (TLX) command transaction diagrams

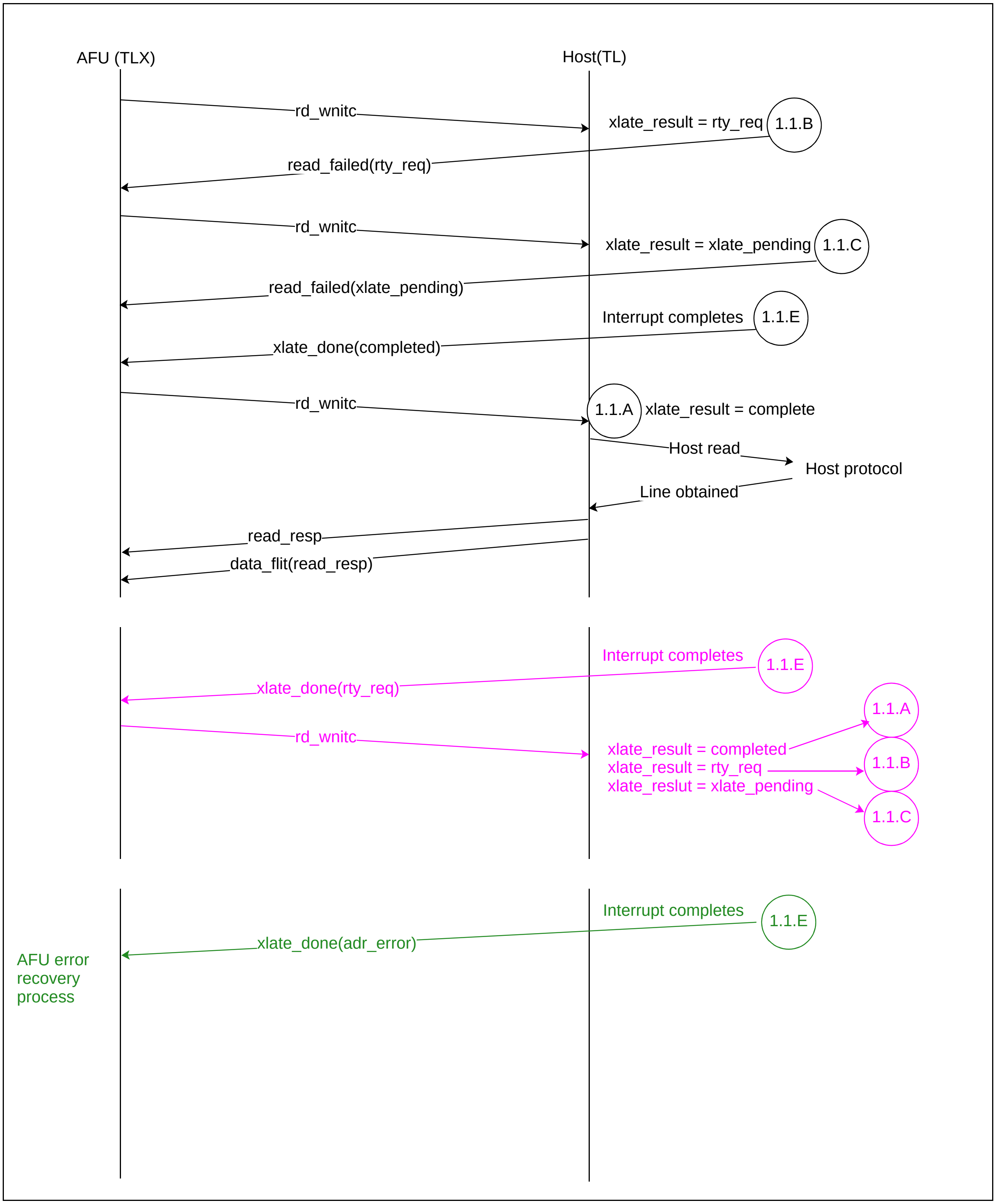
This section contains figures that illustrate AP command flows and TLX and TL interaction.

Rules:

1. Commands received at the TL are not serviced until all data, if any, specified by the AP command has arrived.

A.1 AFU read with no intent to cache; 128 bytes

Figure A-1. TLX and TL interaction: ***rd_wnitc***



A.2 AFU DMA write; 128 bytes

Figure A-2. TLX and TL interaction: ***dma_w*** (Page 1 of 2)

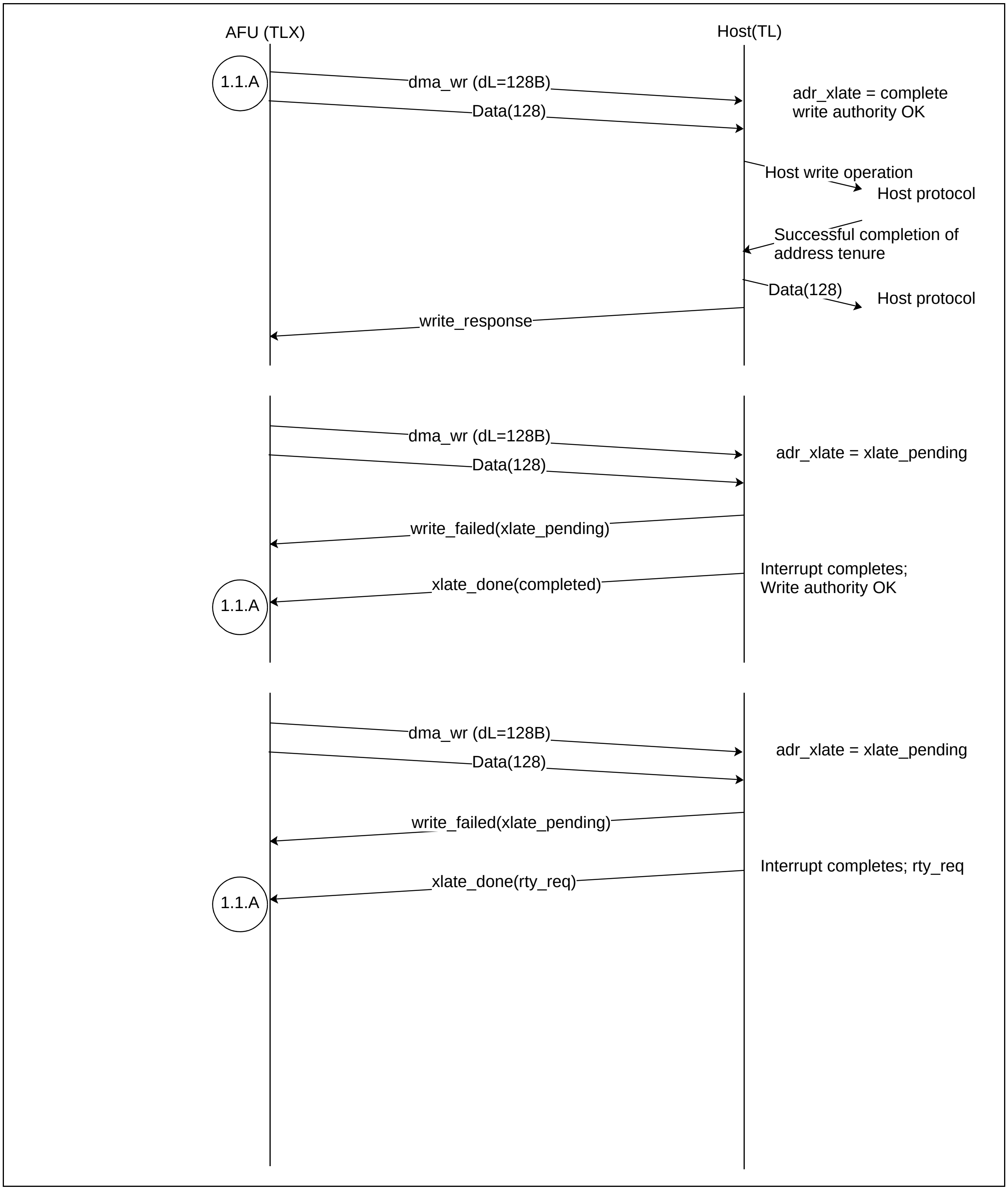
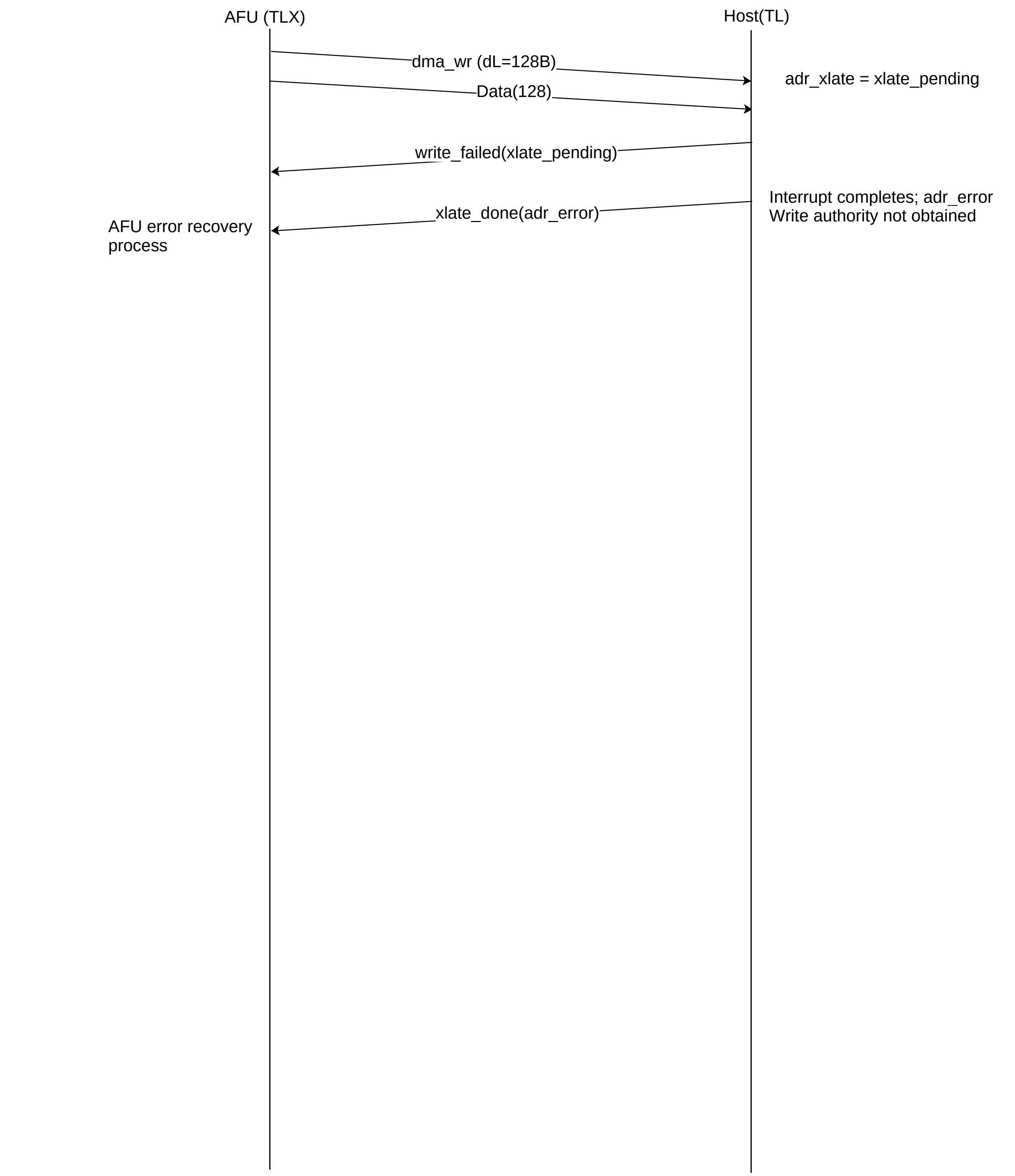
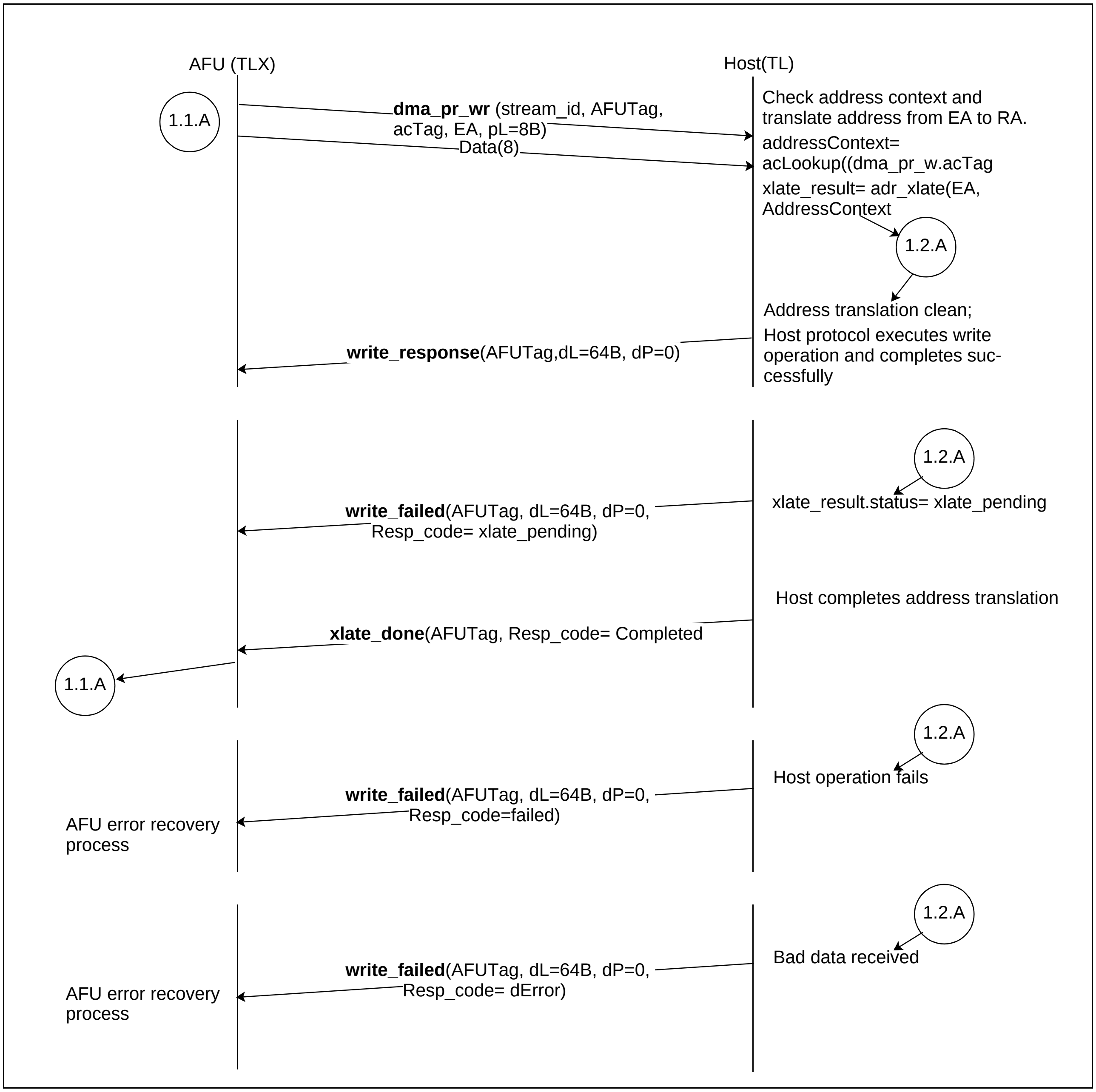


Figure A-2. TLX and TL interaction: ***dma_w*** (Page 2 of 2)


AFU (TLX)
Host(TL)
dma_wr (dL=128B)
adr_xlate = xlate_pending
Data(128)
write_failed(xlate_pending)
Interrupt completes; adr_error
Write authority not obtained
xlate_done(adr_error)
AFU error recovery
process

A.3 AFU DMA partial write; 8 bytes

*Figure A-3. TL and TLX interaction: **dma_pr_w***


AFU (TLX)
Host(TL)
1.1.A
dma_pr_wr (stream_id, AFUTag, acTag, EA, pL=8B)
Data(8)
Check address context and translate address from EA to RA.
addressContext= acLookup((dma_pr_w.acTag
xlate_result= adr_xlate(EA, AddressContext
1.2.A
Address translation clean;
Host protocol executes write operation and completes successfully
write_response(AFUTag,dL=64B, dP=0)
1.2.A
xlate_result.status= xlate_pending
write_failed(AFUTag, dL=64B, dP=0, Resp_code= xlate_pending)
Host completes address translation
xlate_done(AFUTag, Resp_code= Completed
1.1.A
1.2.A
Host operation fails
write_failed(AFUTag, dL=64B, dP=0, Resp_code=failed)
AFU error recovery process
1.2.A
Bad data received
write_failed(AFUTag, dL=64B, dP=0, Resp_code= dError)
AFU error recovery process

Appendix B. CAPP (TL) command transaction diagrams

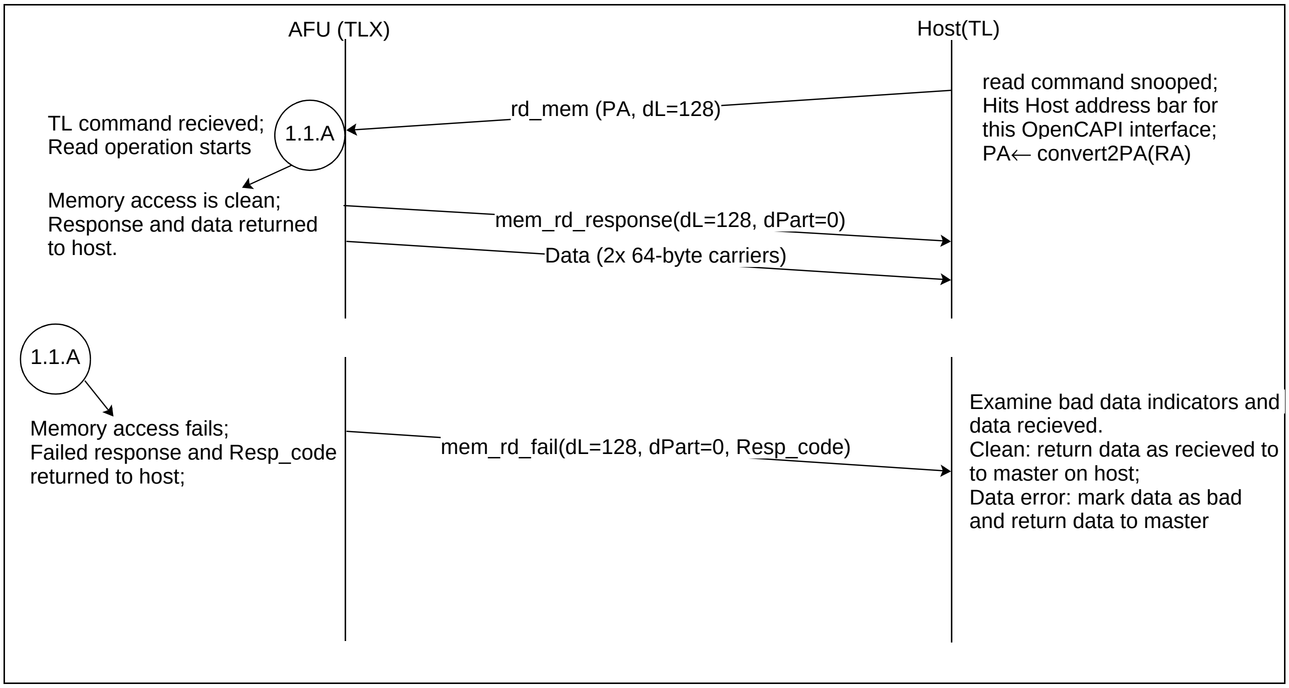
This section contains figures that illustrate CAPP command flows.

Rules:

1. Commands received at the TLX are not serviced until all data, if any, specified by the CAPP command has arrived.

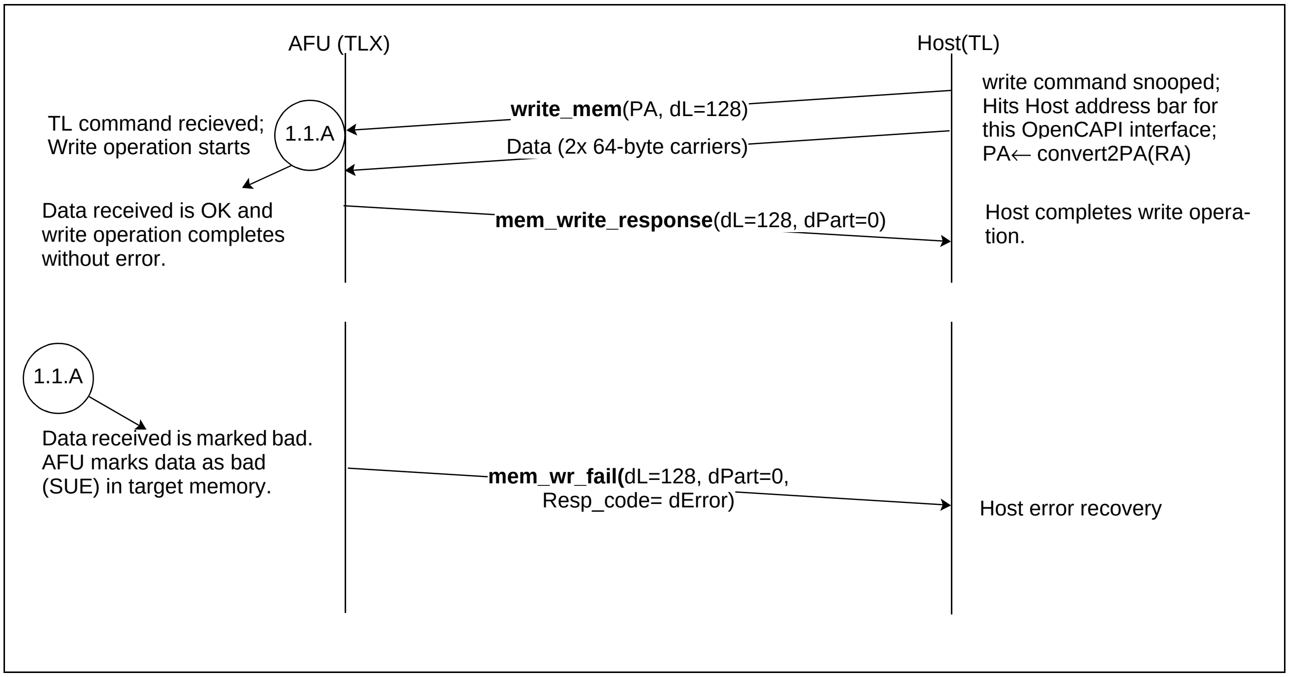
B.1 CAPP memory read; 128 bytes

*Figure B-1. TL and TLX transaction: **rd_mem***


AFU (TLX)
Host(TL)
read command snooped;
Hits Host address bar for
this OpenCAPI interface;
PA← convert2PA(RA)
rd_mem (PA, dL=128)
TL command recieved;
Read operation starts
1.1.A
Memory access is clean;
Response and data returned
to host.
mem_rd_response(dL=128, dPart=0)
Data (2x 64-byte carriers)
1.1.A
Memory access fails;
Failed response and Resp_code
returned to host;
mem_rd_fail(dL=128, dPart=0, Resp_code)
Examine bad data indicators and
data recieved.
Clean: return data as recieved to
to master on host;
Data error: mark data as bad
and return data to master

B.2 CAPP memory write; 128 bytes

*Figure B-2. TL and TLX transaction: **write_mem***


AFU (TLX)
Host(TL)
write command snooped;
Hits Host address bar for
this OpenCAPI interface;
PA← convert2PA(RA)
write_mem(PA, dL=128)
TL command recieved;
Write operation starts
1.1.A
Data (2x 64-byte carriers)
Data received is OK and
write operation completes
without error.
mem_write_response(dL=128, dPart=0)
Host completes write operation.
1.1.A
Data received is marked bad.
AFU marks data as bad
(SUE) in target memory.
mem_wr_fail(dL=128, dPart=0,
Resp_code= dError)
Host error recovery