

# Submission to the p1394.1 committee

David V. James

01Feb99

**This document represents contributions of the author, with the intent of documenting and refining current working group decisions. Due to possible misunderstandings and possible confusions, the contents are *highly preliminary* and *subject to change*.**

The contents of this document include:

- 1) Introduction. Discussion of bridge queues, services, and deadlock avoidance.
- 2) Errors. A proposal for remote-bus acknowledge processing is provided.
- 3) Routing. One set of tables is proposed for routing the following:
  - a) Asynchronous transactions. Routed by their targetId.
  - b) Asynchronous streams. Routed by their sourceId; with local and global variants.
  - c) Clock reference. Routed by the sourceId.
- 4) Formats. Several formats are proposed:
  - a) Portal-generated responses. These packets have an additional responderId field, to identify the actual (as opposed to intended) node where the response was generated.
  - b) Message request/response. These bridge resident message passing locations are defined.
  - c) GASP packets. A few types of GASP packets are proposed:
    - i) Event messages. A form of event message is proposed.
    - ii) Clock reference. Use to adjust the net's clockMasters.

# High performance Serial Bus bridges

## Contacts

This is a *rough draft*, attempting to document the changes proposed in recent working group meetings, based on recollections of working group members. Contact information for this working group activity is as follows:

Web Site: <http://grouper.ieee.org/groups/1394/1/>

WG Chair: Dick Scheel—p1394.1 Chair  
Sony Electronics  
2350 Mission College Blvd, Ste. 982  
Santa Clara, CA 95054  
P: +1.408. 982.5834  
F: +1.408. 982.5899  
C: +1.408.307.1696  
E: dicks@lsi.sel.sony.com

WG editor: Peter Johansson  
Congruent Software, Inc.  
98 Colorado Avenue  
Berkeley, CA 94707  
P: +1.510. 527.3926  
F: +1.510.527.3856  
E: pjohansson@aol.com

This draft editor: David V. James, PhD  
Sony Research Laboratories  
3300 Zanker Road, MS SJ 3D3  
San Jose, CA 95134  
P: +1.408.955.6295  
F: +1.408.955.6060  
H: +1.650.494.0926  
E: davej@lsi.sel.sony.com

## Draft status

### Status 22Jan99

The following questions arose when editing this draft:

- 1) Transaction codes. The following relates to the definition of transaction response codes:
  - a) Response codes. What are the response codes that are returned by the bridge when transactions conclude with a request-ack?
  - b) `ack_tardy` retries. How should the outbound portal respond when an `ack_tardy` is received? How is this behavior different from `ack_complete` or `ack_busy_X`?
  - c) `ack_data_error`. Should this be retried; if so, how many times? Is it really processed like an `ack_busy_X`?
- 2) Multicast routing. Which of the following ways should GASP packets be routed?
  - a) Channel routing. The bridge's channel routing array is used to select the route.
  - b) Source routing. The default GASP channel is hardwired, and that channel is routed based on its `sourceId`. This eliminates the need to calculate yet-another route, but is more restrictive. Should the *sourceId*'s lsb's represent its `phyId` or its `virtualId`?
- 3) Routing tables. Should `r[1023]` be used as a routing enable bit? It works, but is kind of funky.
- 4) Message formats. Should the message start with 16 reserved bits, or should these follow the 48-bit format identifier? Is there any value in standardizing the use of these bits?



## Contents

1.	Introduction.....	9
1.1	Document scope and purpose .....	9
1.2	Bridged systems.....	10
1.2.1	Traditional I/O bridges.....	10
1.2.2	nodeId addresses .....	11
1.2.3	Deadlock free topologies .....	11
1.2.4	Bridge resources.....	12
1.3	Two-portal bridges.....	13
1.3.1	Simple 2-portal bridges.....	13
1.3.2	Generalized 2-portal bridges.....	13
1.4	Bridge queues .....	14
1.4.1	Expected node-to-node transactions .....	14
1.4.2	Asynchronous queues in a bridge .....	14
1.4.3	Isochronous queues in a bridge.....	15
1.4.4	Distinct bridge queues.....	16
1.5	Asynchronous packet routing .....	17
1.5.1	Asynchronous routing checks .....	17
1.5.2	Asynchronous routing tables.....	17
1.6	Isochronous packet routing.....	18
1.6.1	Isochronous routing checks.....	18
1.6.2	Isochronous routing tables .....	18
1.7	Clock synchronization .....	19
2.	References.....	20
3.	Definitions .....	21
3.1	Conformance glossary .....	21
3.2	Technical glossary .....	21
4.	Packet conversions.....	22
4.1	Isochronous packet conversions .....	22
4.1.1	Isochronous CIP packets.....	22
4.1.2	Isochronous CIP packets.....	22
4.1.3	Time stamp changes.....	24
4.2	Asynchronous speed conversions .....	24
4.3	Error and retry handling.....	25
4.3.1	Synthesized responses.....	25
4.3.2	Request acknowledge errors .....	25
4.3.3	Response acknowledge errors.....	26
5.	Virtual IDs .....	27
6.	Packet formats.....	28
6.1	Message formats .....	28
6.2	Bridge generated responses .....	29
6.2.1	Block response .....	29

6.2.2	Write response.....	29
6.2.3	Quadlet read response format .....	30
6.3	Global asynchronous stream packet (GASP) packets.....	31
6.3.1	Global asynchronous stream packet (GASP) format .....	31
6.3.2	Event messages .....	32
6.3.3	Clock adjustment.....	33
7.	Congestion management.....	34
7.1	Receive-queue reservations .....	34
7.1.1	Receive-queue reservations.....	34
7.2	Producer reservation request.....	35
7.3	Consumer reservation filters .....	36
7.3.1	Inbound reservation filter.....	37
8.	Bridge CSRs .....	39
8.1	ERROR_COUNT register .....	39
8.2	ASYNC_SEND routing tables.....	40
8.3	ISOCH_SEND routing tables .....	41
Annex A	(informative) Bibliography .....	43
Annex B	(informative) Alternative designs.....	44
B.1	Coupled bit-mapped routing .....	44
B.1.1	Coupled bit-mapped hardware .....	44
B.1.2	ASYNC_ROUTE tables .....	45
B.2	GASP packet routing .....	46
B.3	Larger N-portal bridges .....	47
B.3.1	N-portal bridge distinctions .....	47
B.3.2	Alternative routing-table structures .....	48
Annex C	(informative) Bridge considerations.....	51
C.1	Potential deadlocks .....	51
C.1.1	Potential queue-dependency deadlocks .....	51
C.1.2	A bridge's potential queue-dependency deadlocks.....	52
Annex D	(normative) Wormhole routing implications.....	53
D.1	Queued send-packet stomping .....	53
D.2	CRC_STOMP definitions .....	53
Annex E	(normative) Code listing.....	55

## Figures

Figure 1— Bridged I/O bridges .....	10
Figure 2— Multiple-bus topologies .....	11
Figure 3— Deadlock free hierarchical routing .....	11
Figure 4— A basic 2-portal bridge .....	13
Figure 5— A generalized 2-portal bridge .....	13
Figure 6— Split-response read transaction .....	14
Figure 7— Split-response read transaction .....	14
Figure 8— Isochronous queues in bridges (minimal jitter) .....	15
Figure 9— Isochronous queues in bridges .....	16
Figure 10— Distinct bridge queues .....	16
Figure 11— Directed transaction routing .....	17
Figure 12— Asynchronous transaction routing-tables .....	17
Figure 13— Directed transaction routing .....	18
Figure 14— Isochronous channel substitution .....	18
Figure 15— Bridged I/O bridges .....	19
Figure 16— CIP packet format .....	23
Figure 17— Remote response synthesis .....	25
Figure 18— Bridge message format .....	28
Figure 19— Block generated read response .....	29
Figure 20— Bridge generated write response .....	29
Figure 21— Bridge generated quadlet read response .....	30
Figure 22— GASP packet format .....	31
Figure 23— Event message format .....	32
Figure 24— Clock adjustment format .....	33
Figure 25— ERROR_COUNT register format .....	39
Figure 26— ASYNC_SEND register formats .....	40
Figure 27— ISOCH_SEND register formats .....	41
Figure B.1— Asynchronous transaction routing-tables .....	44
Figure B.2— ASYNC_ROUTE register format .....	45
Figure B.3— Asynchronous stream routing .....	46
Figure B.4— N-portal bridge .....	47
Figure B.5— N-portal multifunction bridge .....	47
Figure B.6— N-portal switch identifiers .....	48
Figure B.7— Broadcast router checking .....	48
Figure B.8— Source-specified routing .....	49
Figure B.9— Central routing tables .....	49
Figure B.10— Simple dimensional routing .....	50
Figure B.11— Dimensional routing .....	50
Figure C.1— Potential queue-dependency deadlocks, phase1 .....	51
Figure C.2— Potential queue-dependency deadlocks, phase2 .....	51
Figure C.3— Potential deadlocking bridge queues .....	52
Figure C.4— Split-response read transaction .....	52
Figure D.1— Stomped Serial Bus packets .....	53

## Tables

Table 1—Bridge resources .....	12
Table 2—Request subaction processing .....	25
Table 3—Response subaction processing .....	26
Table 4—State transition table for reservation assertion .....	35
Table 5—Consumer reservation filter, design A .....	37
Table 6—Isochronous speed[] values .....	42

# High performance Serial Bus bridges

## 1. Introduction

### 1.1 Document scope and purpose

This document represents several proposals for inclusion in the p1394.1 draft. To reduce later editing difficulties, these proposals have been formatted as proposed for inclusion in the draft. However, the reader should be aware that these proposals are HIGHLY PRELIMINARY and HAVE NOT BEEN approved by the working group.

## 1.2 Bridged systems

### 1.2.1 Traditional I/O bridges

Bridges have traditionally been used to interface I/O buses to the system's high-performance processor/memory bus, as illustrated in figure 1. With such I/O bridges, the CPU can use 4-byte read and write transactions to initiate DMA transfers. When activated, a Serial Bus node's DMA generates split-response read and write transactions; these transactions are forwarded to the intermediate (backbone) Serial Bus. In a similar fashion, most Serial Bus transactions are forwarded through the host-adapter bridge.

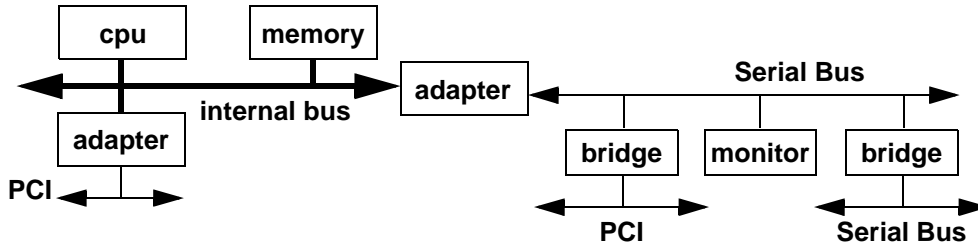


Figure 1—Bridged I/O bridges

Depending on the host system design, the host-adapter bridge may have additional features mandated by differences in bus protocols; for example, the host bus may not directly support isochronous data transfers. Also, the host-adapter bridge could enforce security, by checking and translating incoming transaction addresses, and would often convert uncached I/O transactions into cache-coherent host-bus transaction sequences.

The term *bridge* is used when there is a direct mapping between a majority of the system bus and I/O bus transactions. I/O bridges have several purposes:

- 1) Flexible. Bridges allow different types of buses to be attached.
- 2) Extendable. The number of attached devices can exceed the capacity of one bus.
- 3) Isolating. I/O bridges can partially isolate the processor/memory bus from bus and device failures.

Interfaces to other communication channels, that have no read/write transaction equivalents (such as Ethernet or ATM), are called *controllers*; and typically mandate increased levels of I/O driver software involvement. Capabilities of these controllers, as well as the distinctions between different types of them, are beyond the scope of this standard.

This standard focuses Serial Bus-to-Serial Bus bridges, although many of the results are expected to be applicable to bridges between similar split-response buses conforming to the CSR Architecture.

### 1.2.2 nodeId addresses

Serial Bus conforms to the CSR Architecture and has chosen to use the same 64-bit addressing architecture. Within this 64-bit address, the most-significant 16 bits (the *nodeId*) specify a target-node address; the least-significant 48 bits (the *offset*) specify a location within that node. Based on that fixed-address partitioning, only the most significant 16 bits need be checked when routing decisions are made.

On Serial Bus, 16-bit node address is partitioned into two components: a 10-bit *busNumber* and a 6-bit *busOffset*. Distinct *busNumber* values (*a* through *d*) are expected to be assigned to each bus, as illustrated in figure 2. The 6-bit *busOffset* field (0 through 62) specifies the physical location of the node on its bus. The full node address is a concatenation of these two values (for example, *a.0* or *b.2*).

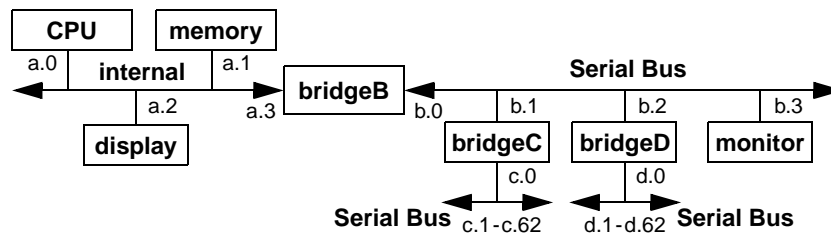


Figure 2—Multiple-bus topologies

In this example, we have assumed that HostBusA has a similar 64-bit fixed addressing convention and (because bridgeB has no address translation/protection capabilities) these addresses have been made visible to devices on the attached I/O buses.

In this environment, each node has a distinct 64-bit address, the most-significant 10-bit portion of that address is used by bridges when making their asynchronous-transaction routing decisions, and the more-significant 16-bit portion of that address is used by nodes when making their acceptance decision. Bridges route asynchronous packets by using the 10-bit *busId* to select 1-of-1024 routing bits; bit values of 0 and 1 indicate the packet ignored or accepted respectively. These routing bits are normally unchanged unless the portal loses power or the net topology changes.

### 1.2.3 Deadlock free topologies

The simplest way to create a set of deadlock-free routing tables is to configure an arbitrary topology into an spanning tree, as illustrated in figure 3. This and many other topologies can be supported by the bridge's bit-mapped tables. After node and bridge characteristics are known, the system may be reconfigured into a more-appropriate operational topology.

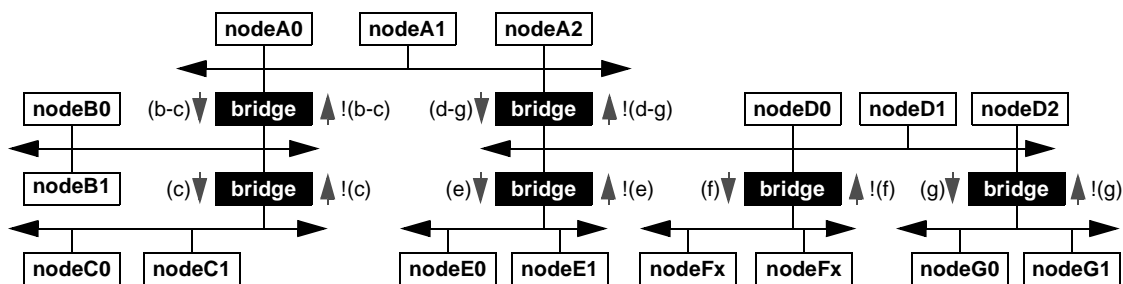


Figure 3—Deadlock free hierarchical routing

### 1.2.4 Bridge resources

Bridging strategies rely on the bus manager protocols, to ensure that nodes with inconsistent context (including bus bridge ports) remain disabled after the bus reset completes. Bridges are also responsible for informing the primary portal when the bus state may have changed.

Bridge provided hardware resources support the capabilities that are listed in table 1 and described further in following subclauses.

**Table 1—Bridge resources**

Resource	Instances	Row	Description
Asynchronous packet queues	request subaction queue	1	Separate request and response queues, to avoid queue-dependency deadlocks
	response subaction queue	2	
Isochronous packet queues	isochronous queue	3	For forwarding isochronous traffic
	cycle start queue	4	For forwarding cycle start calibrations
Routing tables	request/response routing tables	5	Minimum routing table requirement
	isochronous channel selections	6	(plug control registers?)
Message passing locations	Initialization support	7	Establishes bus topology
	Isochronous resource allocation	8	Allocation of isoch channels and bandwidth

**Row 1:** The request queue is needed to hold asynchronous request subaction packets.

**Row 2:** A separate response queue is needed to hold asynchronous request subaction packets. Separate request and response queues are needed to avoid deadlock (see 1.4.2 and C.1).

**Row 3:** The isochronous queue is needed to hold time-sensitive isochronous packets. The isochronous queues are typically several times larger than the maximum Serial Bus packet (see 1.4.3).

**Row 4:** The cycleTime information is effectively queued at incoming portals and regenerated at outgoing portals.

**Row 5:** A set of routing tables is necessary to route asynchronous requests and responses.

**Row 6:** Routing tables is necessary to route isochronous requests and responses.

**Row 7:** Message passing resources are used to establish the bus bridge topology.

**Row 8:** Message passing resources are used to allocate isochronous channels and bandwidth.

### 1.3 Two-portal bridges

This subclause focuses on the use of simple 2-portal bridges, since these are the simplest to construct and can be used to construct nearly arbitrary multiple-bus topologies. In the future, however, the concepts developed for 2-portal bridges are expected to be readily extended to N-portal switches, as described in B.3.

#### 1.3.1 Simple 2-portal bridges

The simplest type of bridge has two portals, connected to busA and busB as illustrated in figure 4. In the context of this standard, this is assumed to be a Serial Bus-to-Serial Bus bridge.

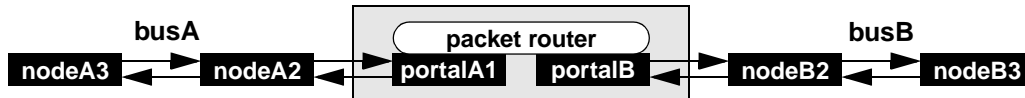


Figure 4—A basic 2-portal bridge

#### 1.3.2 Generalized 2-portal bridges

A generalization of a 2-portal bridge may include other components in the bridge, such as a CPU and memory, as illustrated in figure 5. Co-locating other units on the bridge, rather than forcing these to be attached to one of the attached buses, may be done to improve the system's fault tolerance and performance characteristics.

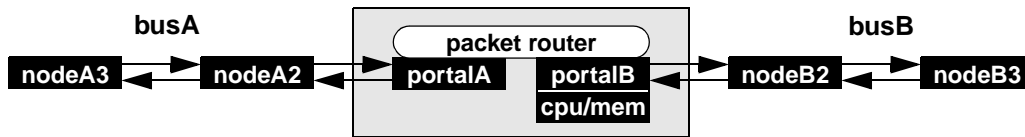


Figure 5—A generalized 2-portal bridge

## 1.4 Bridge queues

### 1.4.1 Expected node-to-node transactions

Asynchronous traffic consists of request and response subactions, which are normally placed into different queues. As an example, a memory read transaction is initiated by a DMA-capable disk controller (the requester) to fetch data from memory (the responder), as illustrated in figure 6. The request sends the address and a transaction-code value to memory; the response returns the data and status from memory. Subactions have distinct *tLabel* values, to affiliate the returned response with the currently-active transaction(s), allowing the subactions to be reordered during their transmission and processing phases.

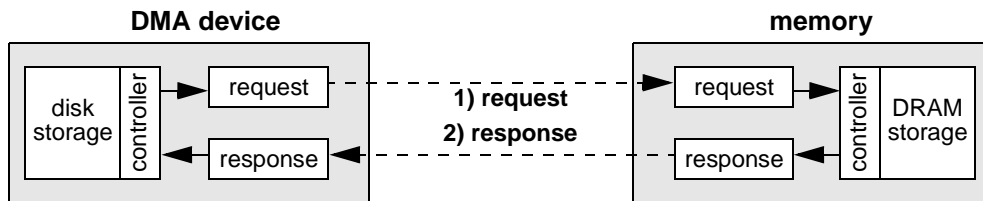


Figure 6—Split-response read transaction

### 1.4.2 Asynchronous queues in a bridge

In larger systems, the requester and responder might be on opposite sides of a bus bridge, as illustrated in figure 7. The bridge acts as an intermediate agent. The bridge accepts (1a) a DMA request and forwards this request (1b) to the memory's bus. Similarly, the bridge accepts (2a) the memory's response for forwarding (2b) to the DMA-bus. If the remote bus is free, the subaction can be pipelined through the bridges (although an *ack\_pending* is still returned); if the remote bus is busy, the subaction is queued in the bridge. Bridges are expected to be symmetric, in that a distinct set of queues (shown in black) is provided for cross-bridge transfers that are initiated from the other bus.

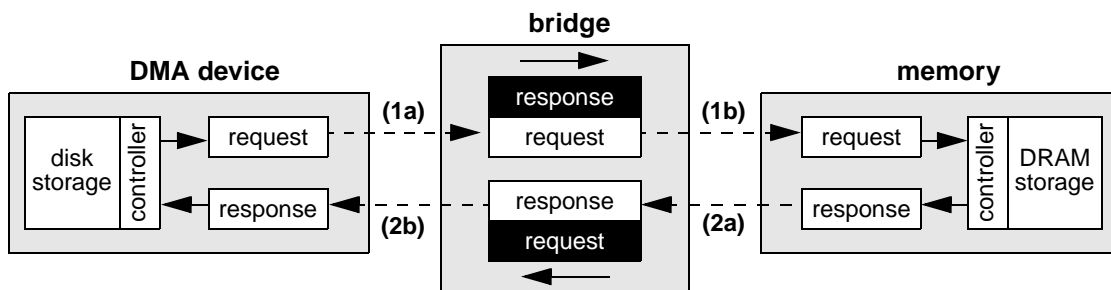


Figure 7—Split-response read transaction

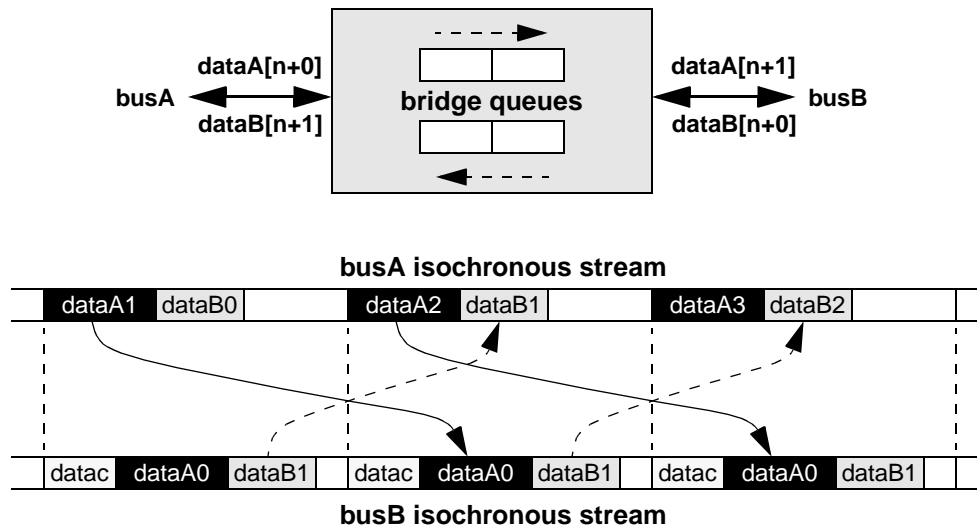
Separate functionally independent bridge queues (one for requests and one for responses) are required to avoid deadlocks. Distinct storage may be preallocated to each queue. Alternatively, these queues may use a common pool of storage locations, if sufficient space is provided and reservation schemes ensure the availability of one sufficiently-sized queue entry for each queue type.

There are no between-queue ordering constraints and subactions from distinct queues or subactions within the same queue may be processed in any order.

### 1.4.3 Isochronous queues in a bridge

Isochronous traffic consists involves a transfer of time-critical data once every 125  $\mu$ s. To maintain the constant isochronous-traffic delivery rate, bridges delays isochronous traffic by a fixed integer number of isochronous cycles. The purpose of this delay is to compensate for the variable arbitration-time related transmission delays in local nodes. The intent is not to minimize the transmission delays, but to limit the cumulative jitter between the packet's worst-case and best-case delivery times. The isochronous packets with long arbitration delays on busA sometimes have a shorter busA-to-busB bridge delay, so the bridge effectively introduces a constant zero-jitter delay.

To simplify synchronized data transfers, all buses effectively operate with the same isochronous cycle clock and bridges introduce a fixed delay. The nominal value for isochronous transfer delay is an integer number of isochronous cycles. With minimal arbitration-jitter assumption, bridges would only have to introduce one cycle of delay, while allowing transfers to be placed at different offsets within each isochronous cycle, as illustrated in figure 8.



**Figure 8—Isochronous queues in bridges (minimal jitter)**

NOTE—Based on previous working group discussions, this figures should show a phase difference between the isochronous cycles on the two buses. If future discussions can provide a rationale for this decision, the rationale will be included and an phase offset will be illustrated. Node that the offset between cycles has an effect on the isochormous storage and perceived delays.

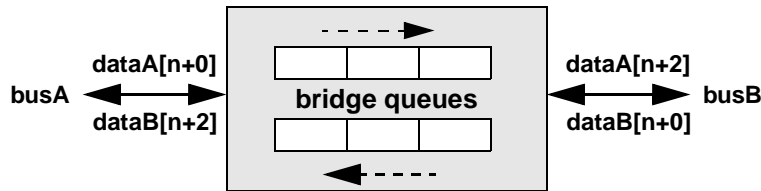
The reason for the 1-cycle through-bridge delay is best illustrated by the dataBx isochronous traffic that is routed from busB-to-busA. The scheduled cycle-offset time on busA is before the scheduled cycle-offset time on busB. The bridge must introduce a partial cycle delay, allowing the scheduled isochronous-cycle offset to be moved forward.

The reason for the 2 cycles of isochronous storage in the bridge is best illustrated by the dataAx isochronous traffic that is routed from busA-to-busB. The scheduled cycle-offset time on busA is after the scheduled cycle-offset time on busB. If a single (rather than double) buffer were to be used, the incoming data from busA could overwrite portions of the previously-queued data destined for busB.

The dataAx transfer is a special case that could (in theory) involve only a fraction-of-one-cycle delay while passing through the bridge. In reality, such latency optimizations are impractical, since 1) the correctness of

such optimizations is load dependent and 2) isochronous transfers are independently scheduled on busA and busB.

Transmission jitter can delay an node's isochronous transmission, from the intended start of cycle[n] into the middle of the following cycle, cycle[n+1]. To compensate for this jitter, practical bridges are expected to introduce at least two cycles of isochronous delay in their busA-to-busB isochronous traffic, as illustrated in figure 9.

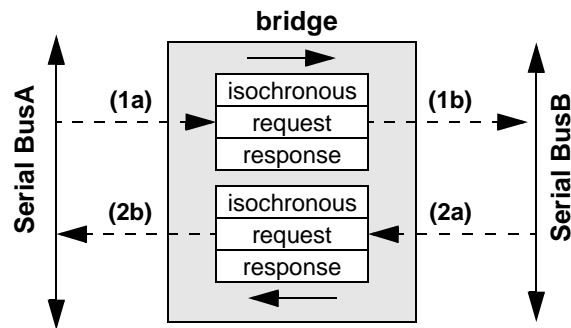


**Figure 9—Isochronous queues in bridges**

For isochronous-transmission delay-jitter budgets between 0.0 and 1.0 cycles, 3-to-4 cycles of isochronous data storage are required respectively.

#### 1.4.4 Distinct bridge queues

Bridges that support the forwarding of isochronous as well as asynchronous traffic have three distinct queues in each direction, as illustrated in figure 10.



**Figure 10—Distinct bridge queues**

Implementations can share physical buffer space, rather than implementing six physical FIFOs, subject to the following constraints:

- 1) Duplex. The busA-to-busB and busB-to-busA queues operate independently.
- 2) Responsive. Acceptance or forwarding of responses is never blocked by queued requests.
- 3) Decoupled. The forwarding of asynchronous requests is not blocked by isochronous requests, and vice-versa.

Although requests and responses should be processed independently, implementations may block a request behind a previously queued response, without generating potential deadlocks.

## 1.5 Asynchronous packet routing

### 1.5.1 Asynchronous routing checks

Directed-asynchronous routing decisions are made by checking the *destination\_ID* addresses of passing-through packets, as illustrated in figure 11. Accepted packets are directly routed to the bridges's opposing port.

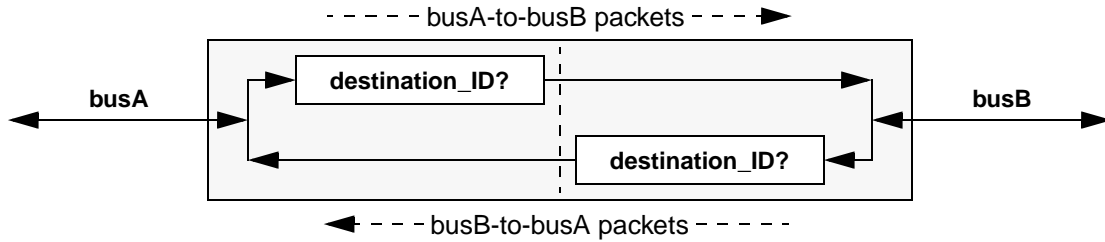


Figure 11—Directed transaction routing

### 1.5.2 Asynchronous routing tables

A simple flexible consumer-routing table structure is assumed, based on the 10 most-significant bits of the request or response packet's *destination\_ID* address, as illustrated in figure 12. This more significant portion of the *nodeId* selects one-of-1024 routing bits and that bit specifies the acceptance condition.

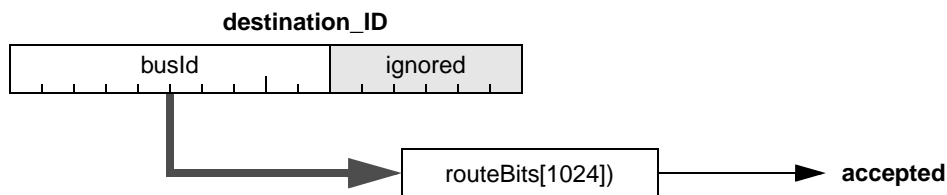


Figure 12—Asynchronous transaction routing-tables

Although less expensive bit-map designs have been proposed (see B.1), this more flexible 1024-bit design is currently viewed to be sufficiently inexpensive to mandate its use in all Serial Bus bridges.

## 1.6 Isochronous packet routing

### 1.6.1 Isochronous routing checks

Isochronous routing decisions are made by checking the isochronous packet's channel number, as illustrated in figure 11. Accepted packets are converted and retransmitted on the adjacent bus. All isochronous packets can have newly-assigned channel numbers and speeds; CIP packets also have substituted *sid* and time-stamp fields (see xx).

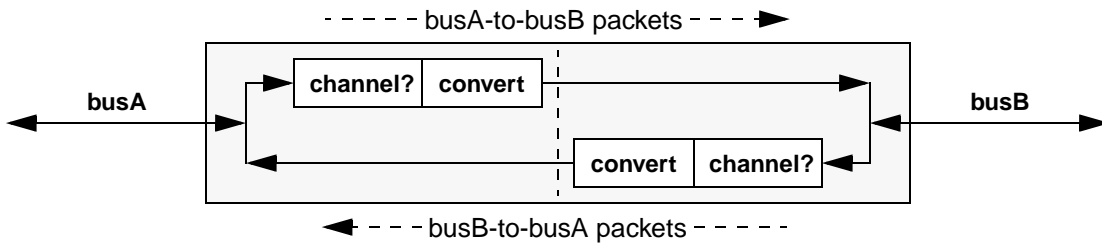


Figure 13—Directed transaction routing

### 1.6.2 Isochronous routing tables

Isochronous routing tables are based on the 6-bit channel number in isochronous packets, which selects one of 64 map entries, as illustrated in figure 14. The largest invalid speed value ( $F_{16}$ ) indicates the isochronous packet should not be accepted.

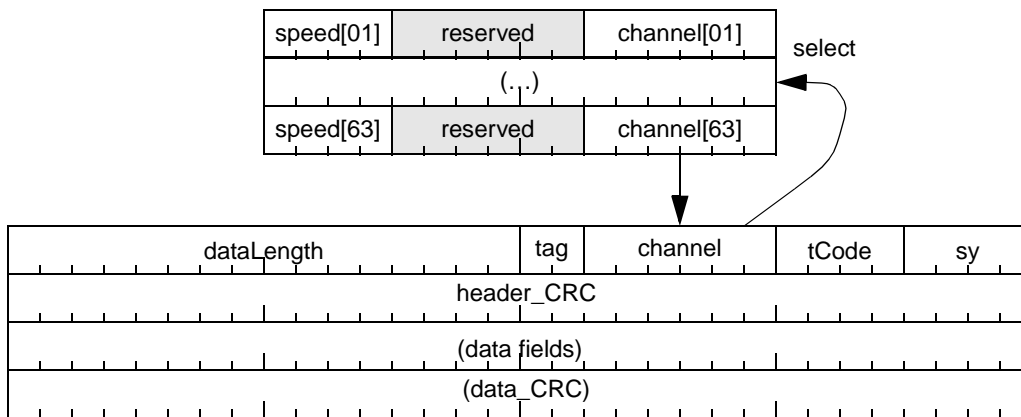


Figure 14—Isochronous channel substitution

## 1.7 Clock synchronization

To simplify interactions, all isochronous-capable nodes are expected to have synchronized timeOfDay clocks. The timeOfDay clock reference is generated one node, called the netClockMaster, and distributed to other nodes, as illustrated by the dotted lines in figure 15.

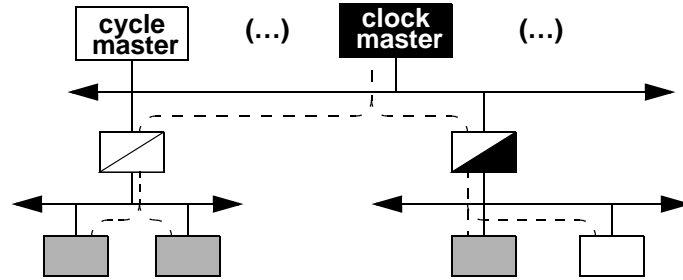


Figure 15—Bridged I/O bridges

Distribution of the clock reference is performed as follows. On the root bus, the cycleMaster generates cycleStart packets, approximately once every 125 $\mu$ s. The clockMaster measures the differences between the cycleMaster’s clock and its reference clock, and reports the observed errors in a special clockSync isochronous packet. On the root bus, this allows the cycleMaster to become synchronized with the clockMaster reference.

On adjacent buses, an outbound bridge portal may assume the roles of the cycleMaster and clockMaster nodes, as shown on the bottom left bus in figure 15. The cycleMaster may sometimes be different from the clockMaster, as shown on the bottom right bus in figure 15. In this case, the remote bus uses the previously discussed protocols to synchronize the cycleMaster with the (bus bridge portal) clockMaster.

To minimize special routing requirements, the clockSync packet is “routed” based on its *sourceId* address. In this case, the packet itself is not routed, but the observed bus-A time reference is redistributed to bus-B (or ignored) based on source-routing tables.

The selection of the clockMaster node is based on the clockMaster’s *nodeId* and a software-settable *preferred* bit, which are included in distributed clockSync packets. The clockMaster selection protocols are simple and deterministic, as follows:

- 1) **Contention.** In the absence of higher precedence clockSync packets, every clockMaster begins generating clockSync packets. To conserve bandwidth, no more than two clockSync packets should be sent in each isochronous cycle.
- 2) **Resolution.** In the presence of a preferred clockMaster source, other clockMaster nodes immediately stop generating clockSync packets.

## 2. References

The following standards contain provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

ANSI/ISO 9899-1990, Programming Language—C.<sup>1,2</sup>

ISO/IEC 13213: 1994 [ANSI/IEEE Std 1212, 1994 Edition], Information Technology—Microprocessor systems—Control and Status Register (CSR) Architecture for Microcomputer Buses.<sup>2</sup>

---

<sup>1</sup>Replaces ANSI X3.159-1989.

<sup>2</sup>ISO/IEC [ANSI/IEEE] publications are available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA or from International Electrotechnical Commission, 3 rue de Varembe, Case Postale 131, CH 1211, Genève 20, Switzerland/Suisse.

### 3. Definitions

#### 3.1 Conformance glossary

#### 3.2 Technical glossary

The following terms are used in this standard:

**3.2.1 CIP.** An acronym for “common isochronous packet”.

**3.2.2 common isochronous packet.** An isochronous packet that has a standard format defined by 61883.

**3.2.3 GASP.** An acronym for “global asynchronous stream packet”.

**3.2.4 global asynchronous stream packet.** A standard asynchronous stream packet format, which supplies the sourceId of the requester and a 48-bit packet format identifier.

**3.2.5 inbound portal.** The bridge portal functionality that eavesdrops and detects an asynchronous request/response packets to be forwarded to another bus.

**3.2.6 outbound portal.** The bridge portal functionality that forwards a previously accepted asynchronous request/response packets to its attache bus.

## 4. Packet conversions

### 4.1 Isochronous packet conversions

#### 4.1.1 Isochronous CIP packets

The outbound portal is responsible for filtering and transforming isochronous data before its transmission. Filtering involves selective discarding of isochronous packets based on the channel number of the isochronous packet, as observed on the inbound portal. After isochronous packets have been filtered, the outbound portal shall transform isochronous packets in the following ways:

- 1) Directed decode. When the target bus is reached, a directed isochronous packet is converted into a local isochronous packet. The 10 more significant bits of the targetId are discarded and the 6 less significant bits specify the local channel number.
- 2) If the packet is specified to have a CIP header, then the following transformations shall also be applied:
  - a) Source ID. The 6-bit sid (source identifier) field shall be set to a the phy\_ID of the bridge, so the listeners can know which node sourced the isochronous packet.
  - b) Time stamps. Time stamps shall be adjusted to account for the apparent isochronous delay between the inbound and outbound portals. Time stamps may be located in two places:
    - i) CIP header. A 4-bit time stamp value is sometimes located in the CIP header.
    - ii) Source packet header. A 13-bit time stamp value is sometimes located in the source packet header; the header location can be derived by evaluation of fields in the CIP header.

To better understand the CIP packet transformations, the CIP-packet format is illustrated and the transformations are described in the following subclauses. Reference C code (annex E) more formally defines the required CIP packet transformations.

#### 4.1.2 Isochronous CIP packets

Isochronous data packets may conform to a common isochronous packet (CIP) format, as defined by IEC 61883/FIS. The presence of a CIP format is indicated by a tag=1 bit in the Serial Bus isochronous packet header, as illustrated in figure 16.

Each quadlet of the CIP header has an *eah* (end of header) bit. For an  $n$  quadlet CIP header, *eah* shall be zero for quadlets zero through  $n-2$ , inclusive, and *eah* shall be one for quadlet  $n-1$ .

Each quadlet of the CIP-header quadlet also has a *form* (format) bit. The current CIP header formats are defined for *form* values of zero; *form* values of one are reserved for future standardization.

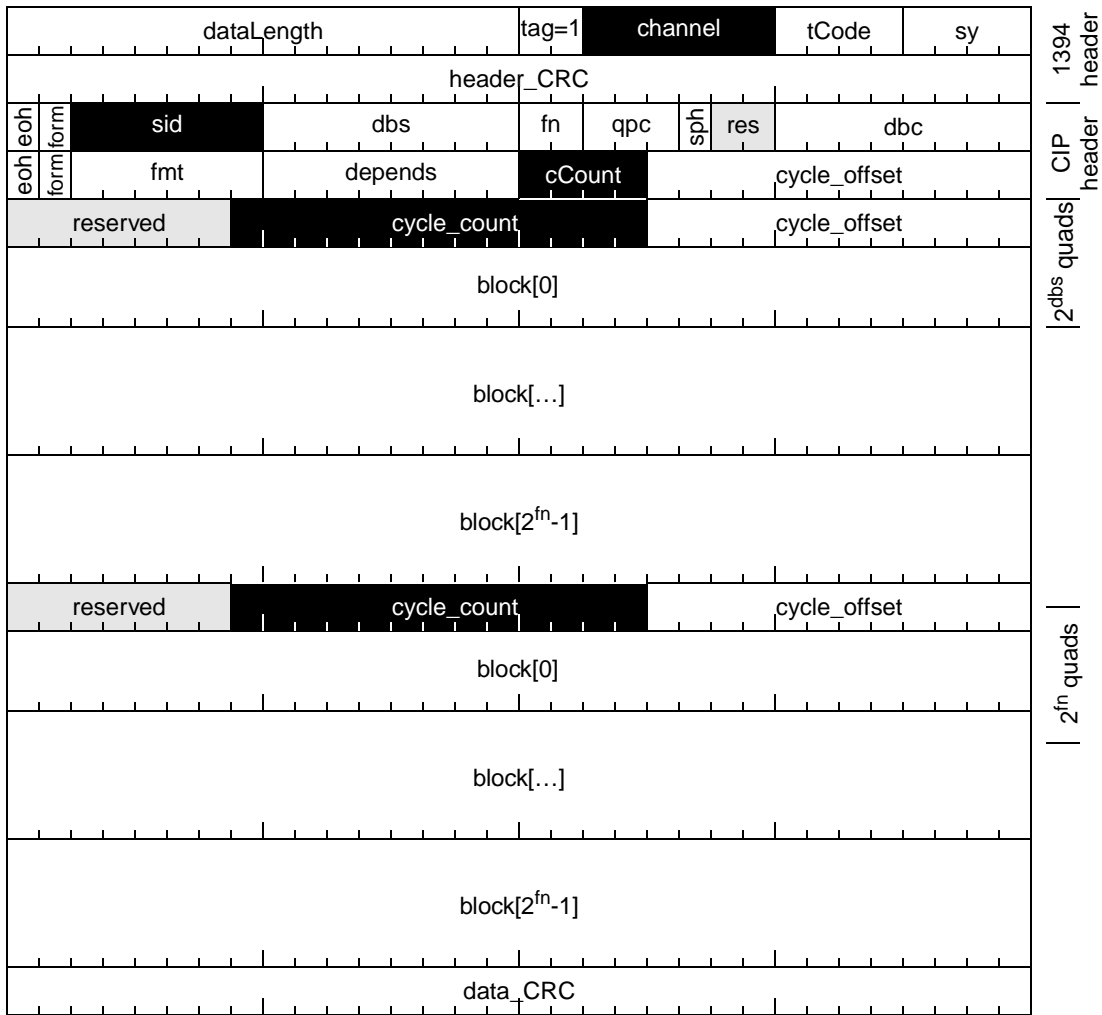
The 6-bit *sid* (source identifier) field normally specifies the Serial Bus physical ID of the source (talker) for the isochronous data. When passing through a bridge, the outgoing port sets this *sid* field to an all 1's value.

The 8-bit *dbz* (data block size) field specifies the size of the application data block(s) that follows the CIP header, as specified in equation 1. More than one data block may be included within a single Serial Bus isochronous packet.

$$\text{blockSize} = (\text{dbz} == 0) ? 256 : \text{dbz}; \quad (1)$$

The 2-bit *fn* (fraction number) field specifies the number of the application data block(s) in the isochronous packet, as specified in equation 2. Possible *blockCount* values include 1, 2, 4, and 8.

$$\text{blockCount} = (1 < \text{fn}); \quad (2)$$



**Figure 16—CIP packet format**

The 3-bit *qpc* (quadlet padding count) is not involved in the packet parameter conversions.

The *sph* (abbreviated as *s*, source packet header) bit shall be 1 if the isochronous source packet begins with a header quadlet that contains the *cycle\_count* and *cycle\_offset* values illustrated in figure 16. Otherwise, the *s* bit shall be zero. The 2-bit *res* field shall be reserved.

The 8-bit *dbc* (data block continuity counter) specifies the sequence number of the isochronous source packet and the sequence number of the data block within the isochronous source packet. The least-significant 8-*fn* bits hold the sequence number of the isochronous source packet. The remaining bits label the first data block that follows the CIP header, and are not involved in the packet parameter conversions.

### 4.1.3 Time stamp changes

The isochronous packet headers identifies the locations of the *cCount* and *cycle\_count* field, one of which must sometimes be adjusted at the outbound portal. Adjustment of the count field compensates for apparent time differences between busA and busB, as specified in equation 1.

$$\text{newCount} = (\text{oldCount} + 8000 + (\text{outTime} - \text{inTime}) + \text{delay}) \% 8000; \quad (1)$$

The *outTime* and *inTime* values represent simultaneous samplings of the cycle-start time on the inbound-portal and outbound-portal buses respectively. The difference is used to compensate for inconsistency errors of timers on adjacent buses.

The *delay* value represents the isochronous pass-through delays of the bridge, measured in units of 125µs. This constant value compensates for the isochronous delays introduced by the bus bridge buffers.

## 4.2 Asynchronous speed conversions

To avoid the complexities of selecting packet size based on its data-transfer paths, the payload portion of remote-bus transactions shall not be greater than 512 bytes in size. This corresponds to the largest packet size supported by the slowest 100Mb link.

By default, outbound portals transmit asynchronous packets at the speed of the slowest portal-to-portal link. In some cases, this may be less than the speed of the connected portals, because the packet may pass through one or more lower speed phys (this is called a speed trap).

TBD—Define how a special bit in the selfId packets helps in this computation, as I think some believe.



### 4.3.3 Response acknowledge errors

After transmitting an outbound asynchronous response packet, the returned acknowledge determines whether the response packet shall be retried, quietly discarded, or discarded after incrementing the portal's error-count value, as specified in table 3.

**Table 3—Response subaction processing**

Request name	Row	Action	Description
ack_complete	1	—	Successful transmission; packet discarded
ack_busy_X ack_busy_A ack_busy_B	2	retry	Subaction retried (see 7.2)
(no acknowledge)	3	errorCount+= 1;	Missing acknowledge error; packet discarded
ack_data_error	4	(retry)	Retries are allowed, until time-of-death timeout
ack_tardy	5	errorCount+= 1;	Should never occur
ack_pending	6	errorCount+= 1;	Illegal acknowledge code; packet discarded
ack_conflict_error	7		
ack_type_error	8		
ack_address_error	9		

## 5. Virtual IDs

TBD—

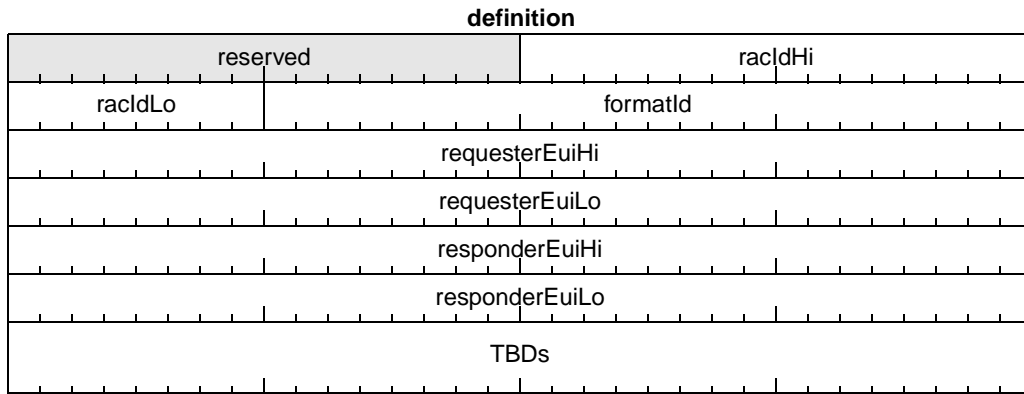
Include what has currently been done.

## 6. Packet formats

### 6.1 Message formats

The bridges communicate with each other by sending 64-byte request and response messages. The request and response packets are normal asynchronous packets, that are directed to the MESSAGE\_REQUEST and MESSAGE\_RESPONSE locations (these locations are defined by the CSR Architecture).

All bridge-related request and response messages have initial identifiers, to distinguish these messages from others, as illustrated in figure 18. The messages also include *requesterEui* and *responderEui*, the EUI-64 values of the requester and responder respectively.



**Figure 18—Bridge message format**

The 16-bit *racIdHi* and 8-bit *racIdLo* values are concatenated to form a 24-bit *racId* value. This *racId* value (see xx) specifies the organization that supplied the *formatId* value.

The 24-bit *formatId* value is concatenated with the *racId* value to form a 48-bit *standardId* value. This *standardId* value corresponds to the standard which defined the remainder of the packet.

The 32-bit *requesterIdHi* and 32-bit *requesterIdLo* values are concatenated to form a 64-bit *requesterId* value. This *requesterId* value shall equal the EUI-64 identifier of the requester.

The 32-bit *responderIdHi* and 32-bit *responderIdLo* values are concatenated to form a 64-bit *responderId* value. This *responderId* value shall equal the EUI-64 identifier of the responder.

NOTE—If the target’s EUI-64 differs from the message-specified value, the message is assumed to have been misdirected and is therefore discarded.

## 6.2 Bridge generated responses

A bridge sometimes generates a response on behalf of the responder. When this occurs, the bridge places its *nodeId* in the *responderId* field of the response, as illustrated in the following subclauses. Returning the *responderId* value is intended to simplify diagnostic software, by identifying where errors are located.

### 6.2.1 Block response

The format of most bridge-generated block-read responses packet includes a 16-bit *responder\_ID* field, as illustrated in figure 19. The bridge-generated quadlet-lock and octlet-lock responses shall have the same format.

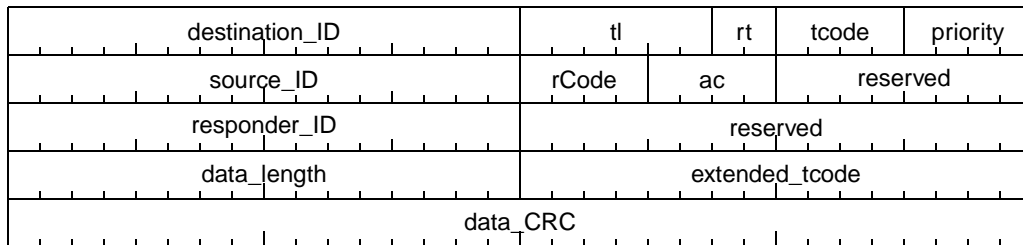


Figure 19—Block generated read response

A distinct bridge generated responses, the 4-bit *aCode* field is nonzero and specified by Serial Bus table 6-13.

For nonzero *aCode* values, the 16-bit *responder\_ID* field is the *nodeId* of the bridge portal where the completion acknowledge was observed.

The remaining fields within the block read response are defined by the Serial Bus standard.

### 6.2.2 Write response

The block-write response packets includes a 16-bit *responder\_ID* field, as illustrated in figure 20. The bridge-generated quadlet-write response shall have the same format.

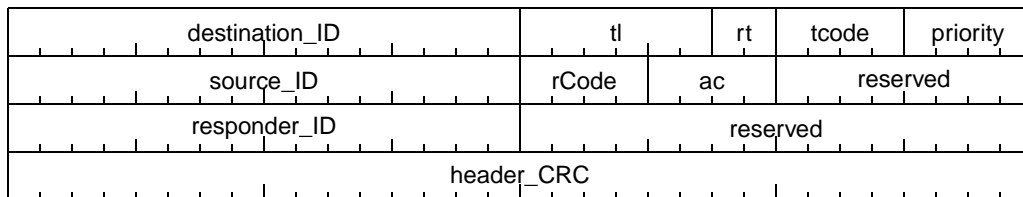


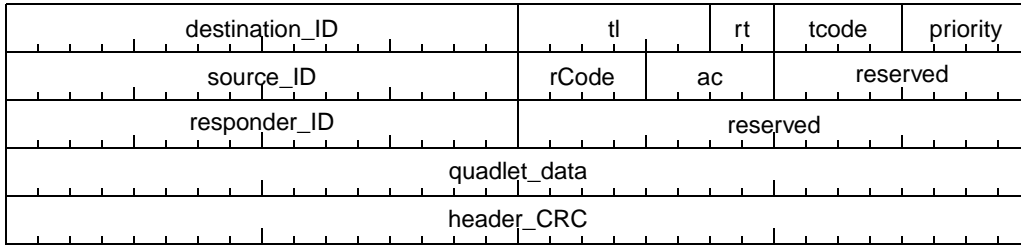
Figure 20—Bridge generated write response

If *rid* is zero, the request was processed by the *destination\_ID*-specified responder and the *responder\_ID* field shall be reserved. If *rid* is 1, the request was generated by a bridge portal and *responder\_ID* equals the *nodeId* of the bridge portal.

The remaining fields within the block write response are defined by the Serial Bus standard.

### 6.2.3 Quadlet read response format

The quadlet read response packet includes a 16-bit *responder\_ID* field, as illustrated in figure 21.



**Figure 21—Bridge generated quadlet read response**

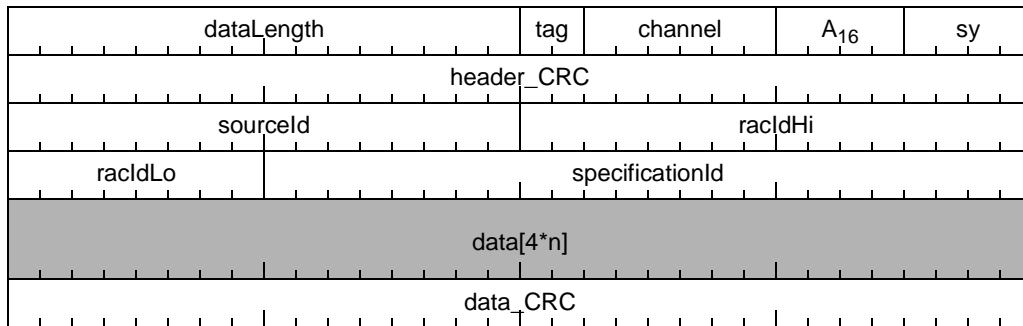
If *rid* is zero, the request was processed by the destination\_ID-specified responder and the *responder\_ID* field shall be reserved. If *rid* is 1, the request was generated by a bridge portal and *responder\_ID* equals the nodeId of the bridge portal.

The remaining fields within the block write response are defined by the Serial Bus standard.

### 6.3 Global asynchronous stream packet (GASP) packets

#### 6.3.1 Global asynchronous stream packet (GASP) format

Bridges are only expected to forward asynchronous stream traffic which has GASP (global asynchronous stream packet) formats. A distinctive tag-field code differentiates the GASP packet from other isochronous traffic, as illustrated in figure 22.



**Figure 22—GASP packet format**

Either local or global broadcasts can be supported, as follows:

- 1) Local broadcast. When the 10 more significant bits of the sourceId are 3FF<sub>16</sub>, a local broadcast is performed. The 6 less significant bits of the sourceId correspond to the producer's phyId address.
- 2) Global broadcast. When the 10 more significant bits of the sourceId are different from 3FF<sub>16</sub>, a global broadcast is performed. The 6 less significant bits of the sourceId correspond to the producer's phyId address.

Several types of GASP packets are defined for bridges, as specified in the following subclauses.

### 6.3.2 Event messages

An event message broadcasts state change information, where that state change information has the format illustrated in figure 23. If two event messages must be merged, a null-valued *source\_ID* value is generated and the event bits *e[i]* shall be OR'd together. The intent is to provide a reliable event notification service, to alleviate the need for excessive numbers of bus resets.

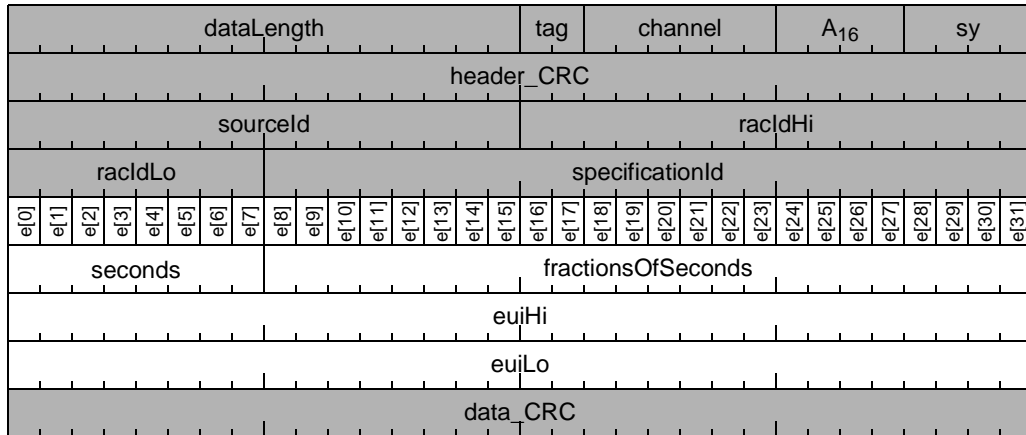


Figure 23—Event message format

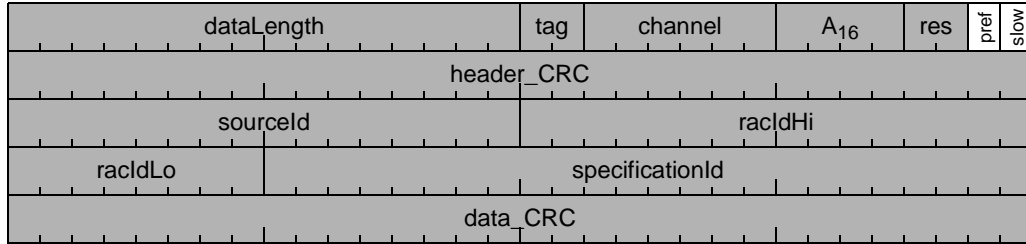
The 32-bit *euiHi* and 32-bit *euiLo* values are concatenated to form a 64-bit *eui* value, which specifies the *eui64* value of the desired target. Although broadcasts of this form are inefficient, since these packets are sent to many nodes other than the target, they are useful when the *eui64* of the target is known and its *nodeId* is not.

If possible, this information would have been transmitted as asynchronous multicast packets. Since these are not supported on Serial Bus, they are transported as asynchronous stream packets. The convention for merging of two events, when insufficient queue space is available, is as follows:

- 1) Precedence. The packet with the oldest time stamp has precedence.
- 2) Source selection. The *source\_ID* is copied from the higher precedence packet.
- 3) Event merging. The event bits, *e[i]*, are calculated by OR'ing the corresponding bits in higher and lower precedence packets.
- 4) EUI-64 merging. If the higher and lower precedence *eui* values are equal, the merged value is selected as either; otherwise, the merged value shall be all ones.

### 6.3.3 Clock adjustment

A clock adjust message indicates whether the clock master should run faster or slower, or (in the packet's absence) whether the nominal clock rate is desired. Only one bit within the GASP packet is used for this purpose, as illustrated in figure 24.



**Figure 24—Clock adjustment format**

The *pref* bit value is prepended with the sourceId value to generate a 17-bit clockSelect value. The clockReference-capable node with the largest clockSelect value becomes the net's clockReference node.

If the *slow* bit value is 0 or 1, the clock frequency for the next cycle shall be one unit slow or one unit fast respectively.

This clockAdjust packet is not directly forwarded to remote packets, but indirectly forwarded in the sense that it affects the behavior of adjacent portals. The remote portals become local clock references and include the *sourceId* value in their clock adjustment packets. This form of source-routing uses the asynchronous routing tables whose deadlock free properties ensure nonlooping clock reference distribution.

## 7. Congestion management

NOTE—This discussion is intended to be incorporated into the p1394b specification. For convenience, and since bus bridges are the most likely mandatory candidate, it has been first documented in this draft.

### 7.1 Receive-queue reservations

#### 7.1.1 Receive-queue reservations

To ensure forward progress, fair arbitration protocols and fair acceptance protocols are necessary. Fair arbitration protocols allow each nodes to transmit packets; fair acceptance protocols ensures that transmitted packets will eventually be accepted, rather than busied. Acceptance protocol fairness is based on reserving future queue space to older sets of requests, where the age of a request is based on the time of its initial retry.

Allocation reservations have timeouts, so that the reservation can be reclaimed in the absence of retries. Reservation timeouts may occur when an active node is reset, when transmission errors corrupt the packet header, or if the packet can't be transmitted before its time-of-death is reached. The minimal interval between retries is specified, to avoid falsely triggering one of these missing-retry timeouts.

For fairness, each producer shall eventually tag its oldest request and its oldest response, with a reservation-requested label. A high performance producer may tag multiple requests (or responses) with reservation-requested labels, if each of these is directed to a different consumer.

The basis of the reservation protocols is as follows:

- 1) Age labels. The producer initially labels its oldest retries with a reservation-requested `retry_1` label; newer requests are labeled with a `retry_X` label.
- 2) Assignment. Reservations are assigned to `retry_1` packets, by returning an `ack_busy_A` or `ack_busy_B` label.
- 3) Servicing. The oldest of the `retry_A/retry_B` packet retries are allowed to consume queue space while other are busied.

## 7.2 Producer reservation request

The producer’s handling of transmission labels is specified by the state machine of table 4. Although not listed in this table, the producer shall retry a previously busied subaction with no more than AC intervening arbitration gaps.

**Table 4—State transition table for reservation assertion**

inputs		Row	results	
old state	ack-returned condition		new state	transmitted send.rt
—	Reset completion	1	DONE	—
DONE	newer packet available	2	SEND	retry_X
SEND	TimeOfDeath()  !AckBusy();	3	DONE	—
	oldest&&ack_busy_X	4	—	retry_1
	oldest&&ack_busy_A	5	—	retry_A
	oldest&&ack_busy_B	6	—	retry_B
	!oldest&&AckBusy();	7	—	retry_X

**Row 1:** Reservation assignments are discarded during a bus reset.

**Row 2:** The packet is initially sent with a retry\_X indication.

**Row 3:** The packet is no longer retried when its time-of-death timeout has been exceeded or when a busy acknowledge (ack\_busy\_X, ack\_busy\_A, or ack\_busyB) is not returned.

**Row 4:** The oldest packet changes from retry\_X for retry\_1 when busied for the first time. The intent is to request a reservation on the next retry.

**Row 5:** The oldest packet inherits the ack\_busy\_A acknowledge, accepting this reservation assignment.

**Row 6:** The oldest packet inherits the ack\_busy\_B acknowledge, accepting this reservation assignment.

**Row 7:** The newer packets continue to retry with their initial retry\_X label.

### 7.3 Consumer reservation filters

Revisions to the inbound busyA/B retry state machine (page 191, 1394-1995) are proposed, for the following reasons:

- 1) No resynchronization. The existing protocols doesn't automatically resync when producer and consumer have inconsistent reservation histories. State machines need to remain wait for a retry-timeout so that any outstanding (but unaccounted for) reservations will be serviced.
- 2) Retry timeouts. The retry timeout for the producer (once every 4 arbitration intervals) is inconsistent with that of the consumer (once every 3 arbitration intervals) based on some interpretations.
- 3) No counts. Its unclear how to extend the current specification to incorporate reservation counts.

To fix these known problems and clarify the definition, new inbound state machines are proposed for p1394a. Both of these are thought to be correct and complete. Option A is preferred by the authors of these proposals (Farrell Ostler and David James), but option B is more consistent with past nomenclature and may therefore be more acceptable to reviewers. We propose to post both to the reflector, to solicit comments from a broader audience.

### 7.3.1 Inbound reservation filter

The inbound reservation filter has two states: USE\_A and USE\_B. The ‘A’ and ‘B’ reservations have precedence when in the USE\_A and USE\_B states respectively, as specified in table 5. Acceptance filters remain in these states for a minimum of four arbitration intervals, so that any outstanding (but unaccounted for) reservations will be serviced.

**Table 5—Consumer reservation filter, design A**

old state	condition	Row	action	ack	new state
—	Reset completion	1	ra=rb=ac=0	—	USE_A
USE_A	Queue() != FULL && send.rt == retry_A	2	Sub(ra,RC)	AckDone()	USE_A
	Queue() != FULL && ra == 0 && send.rt == retry_B	3	Sub(rb,RC)		
	Queue() != FULL && ra == 0 && send.rt == retry_1	4	—		
	Queue() != FULL && ra == 0 && send.rt == retry_X	5	—		
	Queue() == FULL && ra != 0 && send.rt == retry_A	6	ac=0	ack_busy_A	
	Queue() == FULL && ra == 0 && send.rt == retry_A	7	ac=0, Add(ra)	—	
	!(Queue() != FULL && ra == 0) && send.rt == retry_B	8	—	ack_busy_B	
	!(Queue() != FULL && ra == 0) && send.rt == retry_1	9	Add(rb,RC)	—	
	!(Queue() != FULL && ra == 0) && send.rt == retry_X	10	—	ack_busy_X	
	ArbResetGap && ac != AC	11	ac += 1;	—	
	ArbResetGap && ac == AC	12	ac = 1, ra = 0;	—	
USE_B	Queue() != FULL && send.rt == retry_B	13	Sub(rb,RC)	AckDone()	USE_B
	Queue() != FULL && rb == 0 && send.rt == retry_A	14	Sub(ra,RC)		
	Queue() != FULL && rb == 0 && send.rt == retry_1	15	—		
	Queue() != FULL && rb == 0 && send.rt == retry_X	16	—		
	Queue() == FULL && rb != 0 && send.rt == retry_B	17	ac=0	ack_busy_B	
	Queue() == FULL && rb == 0 && send.rt == retry_B	18	ac=0, Add(rb)	—	
	!(Queue() != FULL && rb == 0) && send.rt == retry_A	19	—	ack_busy_A	
	!(Queue() != FULL && rb == 0) && send.rt == retry_1	20	Add(ra,RC)	—	
	!(Queue() != FULL && rb == 0) && send.rt == retry_X	21	—	ack_busy_X	
	ArbResetGap && ac != AC	22	ac += 1;	—	
	ArbResetGap && ac == AC	23	ac = 1, rb = 0;	—	

Notes:

```
#define AC 4 // Reservation timeout interval
#define Add(a,b) (a += (a != b))
#define Sub(a,b) (a -= (a != b && a != 0))
// RC is implementation dependent
```

The *ac* counter counts the number of consecutive arbitration gaps, and is limited by the *AC* value that specifies the producer's worst-case retry delay.

The *ra* counter counts the number of *A* reservation assignments; the *rb* counter counts the number of *B* reservation assignments. The *RC* value, that specifies the size of these counters, is implementation dependent; in the absence of a counter, the state machines shall behave as though *RC* were equal to 1.

**Row 1:** Reservation assignments are discarded during a bus reset.

**Row 2, row 13:** Current reservation-assigned packets are always accepted.

**Row 3, row 14:** Later reservation-assigned packets are accepted, if current reservations have been serviced.

**Row 4, row 15:** Later reservation-requested packets are accepted, if current reservations have been serviced.

**Row 5, row 16:** Reservationless packets are accepted, if current reservations have been serviced.

**Row 6, row 17:** A busied current reservation-assigned packet clears the retry timeout counter.

**Row 7, row 18:** Unexpected current reservation assignments are honored.

**Row 8, row 19:** Later reservations are busied, while current reservations are outstanding.

**Row 9, row 20:** When full or reserved, a reservation-requested packets obtains a later reservation.

**Row 10, row 21:** When full or reserved, a reservationless packets is busied without reservations.

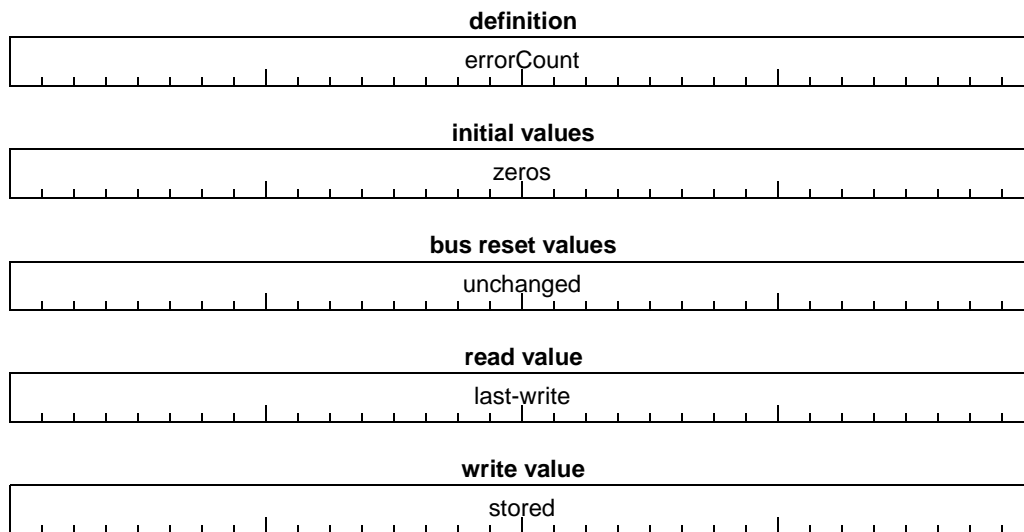
**Row 11, row 22:** Reservation timeout interval is measured in units of arbitration reset gaps.

**Row 12, row 23:** Later reservations become current when the reservation timeout is reached.

## 8. Bridge CSRs

### 8.1 ERROR\_COUNT register

Bridges shall have an ERROR\_COUNT register, located at offset 384 within each portal's CSR space. This register provides access to that portal's errorCount value, as illustrated in figure 25.

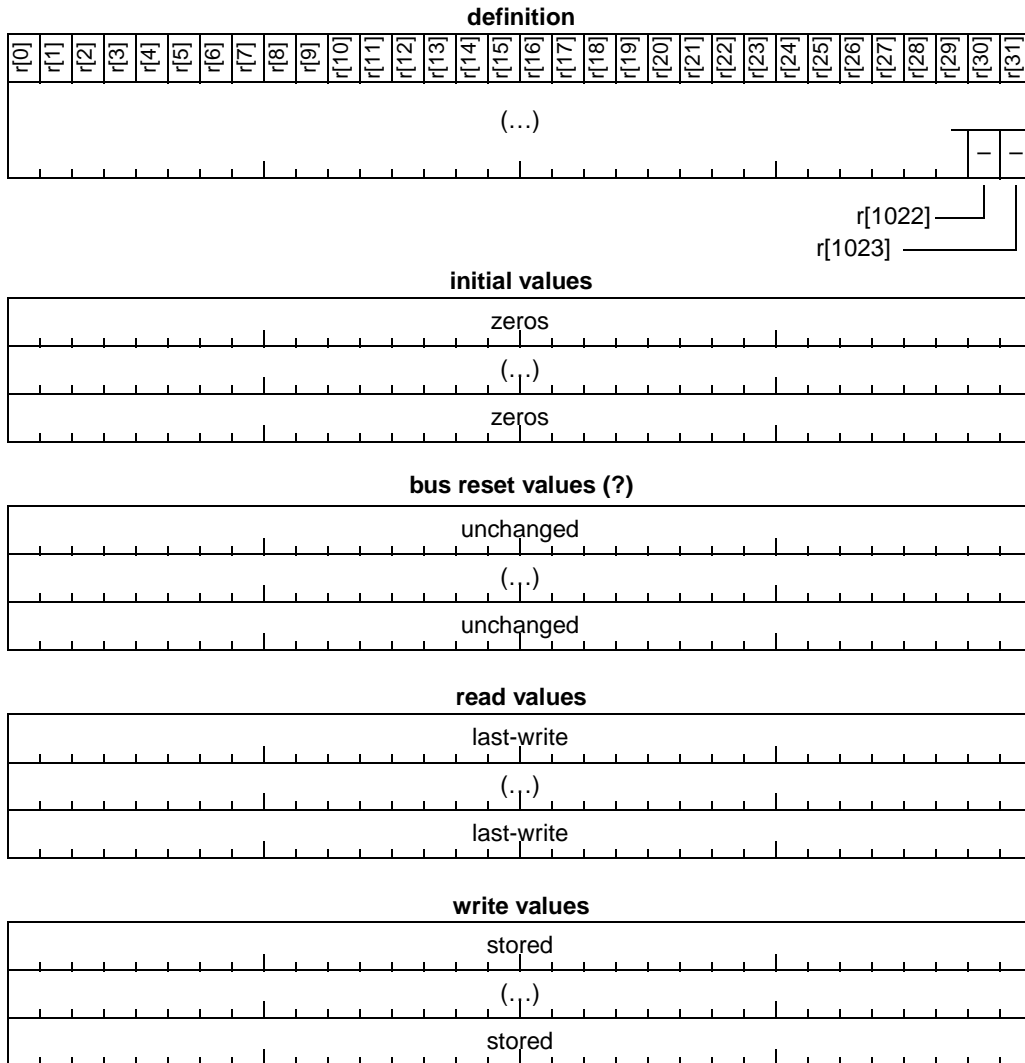


**Figure 25—ERROR\_COUNT register format**

A portal may provide additional registers to describe the cause or circumstances surrounding the error. The definition of such registers is beyond the scope of this standard.

## 8.2 ASYNC\_SEND routing tables

A sophisticated bridge is expected to provide ASYNC\_SEND isochronous routing tables. The directed routing tables use 16 bits to specify how each of the local isochronous channels is forwarded to the remote bus, as illustrated in figure 26.



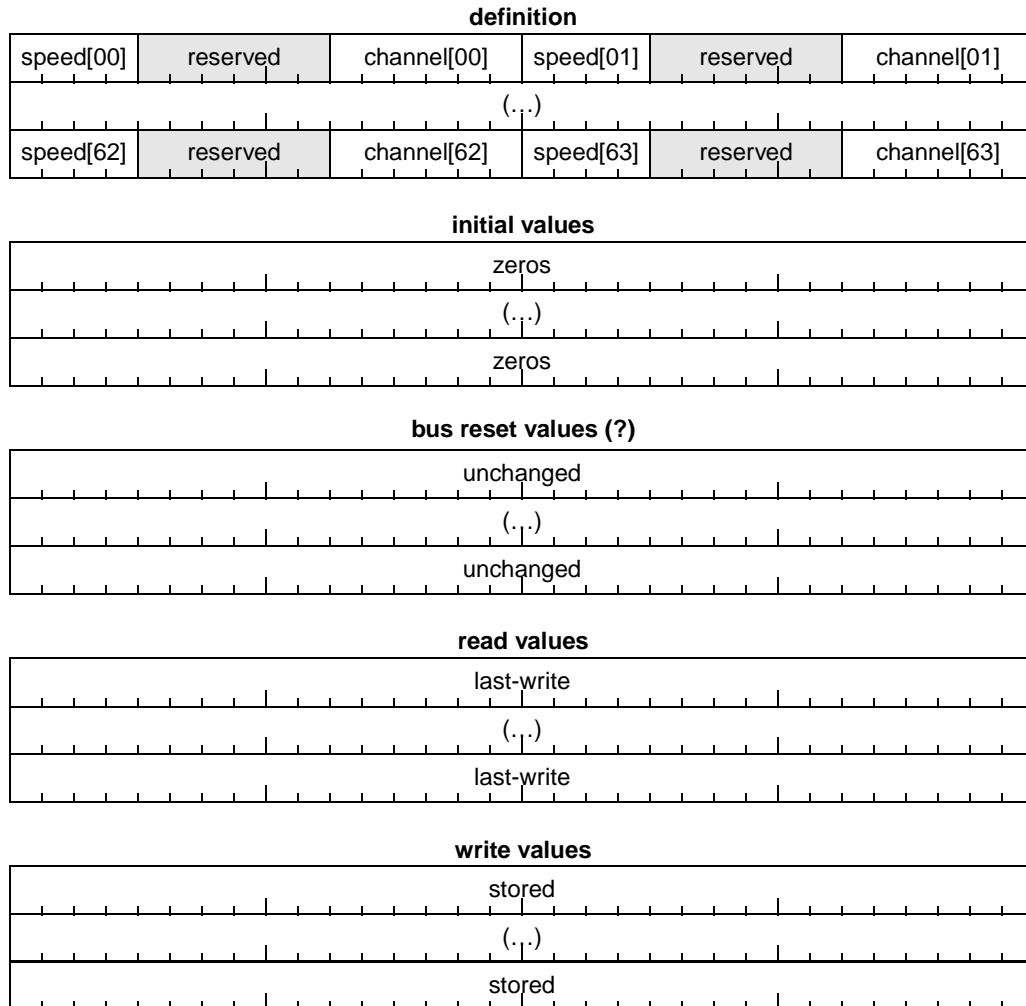
**Figure 26—ASYNC\_SEND register formats**

The acceptance condition for asynchronous packets is based on these routing bits, as follows:

$$\begin{aligned}
 \text{busId} &= \text{packetPtr} \rightarrow \text{sourceId} \gg 6; \\
 \text{accepted} &= \text{busId} \neq 1023 \&\& \text{r}[\text{busId}] \neq 0 \&\& \text{r}[1023] \neq 0;
 \end{aligned}
 \tag{2}$$

### 8.3 ISOCH\_SEND routing tables

A sophisticated bridge is expected to provide ISOCH\_SEND isochronous routing tables. The directed routing tables use 16 bits to specify how each of the local isochronous channels is forwarded to the remote bus, as illustrated in figure 27.



**Figure 27—ISOCH\_SEND register formats**

The meaning of the *speed[i]* fields is specified in table 6.

**Table 6—Isochronous *speed[]* values**

<b>Value</b>	<b>Name</b>	<b>Description</b>
0	NULL	Channel forwarding disabled
1	S100	Forward at S100
2	S200	Forward at S100
3	S400	Forward at S100
4-15	—	Reserved

## **Annexes**

### **Annex A**

(informative)

### **Bibliography**

The documents listed below are useful for understanding this specification.

[B1] ANSI X3.159-1989, Programming Language—C.

## Annex B

(informative)

### Alternative designs

NOTE—The following subclauses document alternative designs that were not selected by this standard. These alternatives are being maintained until the documentation and viability of the selected designs has been verified.

Questions that remain unclear to the author, and are explored further in this annex, include the following:

- 1) Time synchronization. Since we are sending an isochronous packet anyway, to synchronize other clocks with the reference, maybe:
  - a) The full correction value (32-bit fractions of a second) should be sent.
  - b) Time should be globally synchronized.
- 2) Routing tables. Should the coupled bit-mapped routing be used?  
Compared to the full bit-map table, its less expensive yet sufficiently flexible for most applications.

#### B.1 Coupled bit-mapped routing

##### B.1.1 Coupled bit-mapped hardware

NOTE—This bit map routing table has the advantage of being smaller than a full 1024-bit table, but places some restrictions on the routing options that can be supported. If the 1024-bit table continues to be viewed as an insignificant cost, this alternative may disappear in future revisions of this standard.

A simple flexible consumer-routing table structure is assumed, based on the 10 most-significant bits of the request or response packet's *destination\_ID* address, as illustrated in figure B.1. The lower bits of the address select 1-of-32 routing-bit pairs. The higher bits of the address select 1-of-32 usage-bit pairs. For each 32-bus block, the two usage bits specify whether the address block is never accepted, is accepted based on 1-of-2 routing-bit values, or is always accepted.

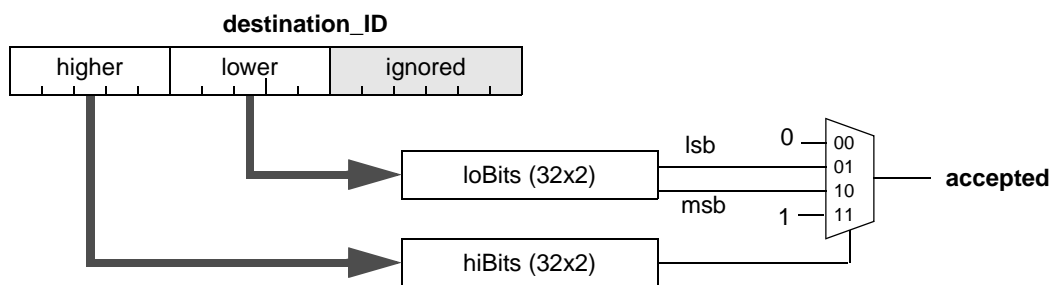
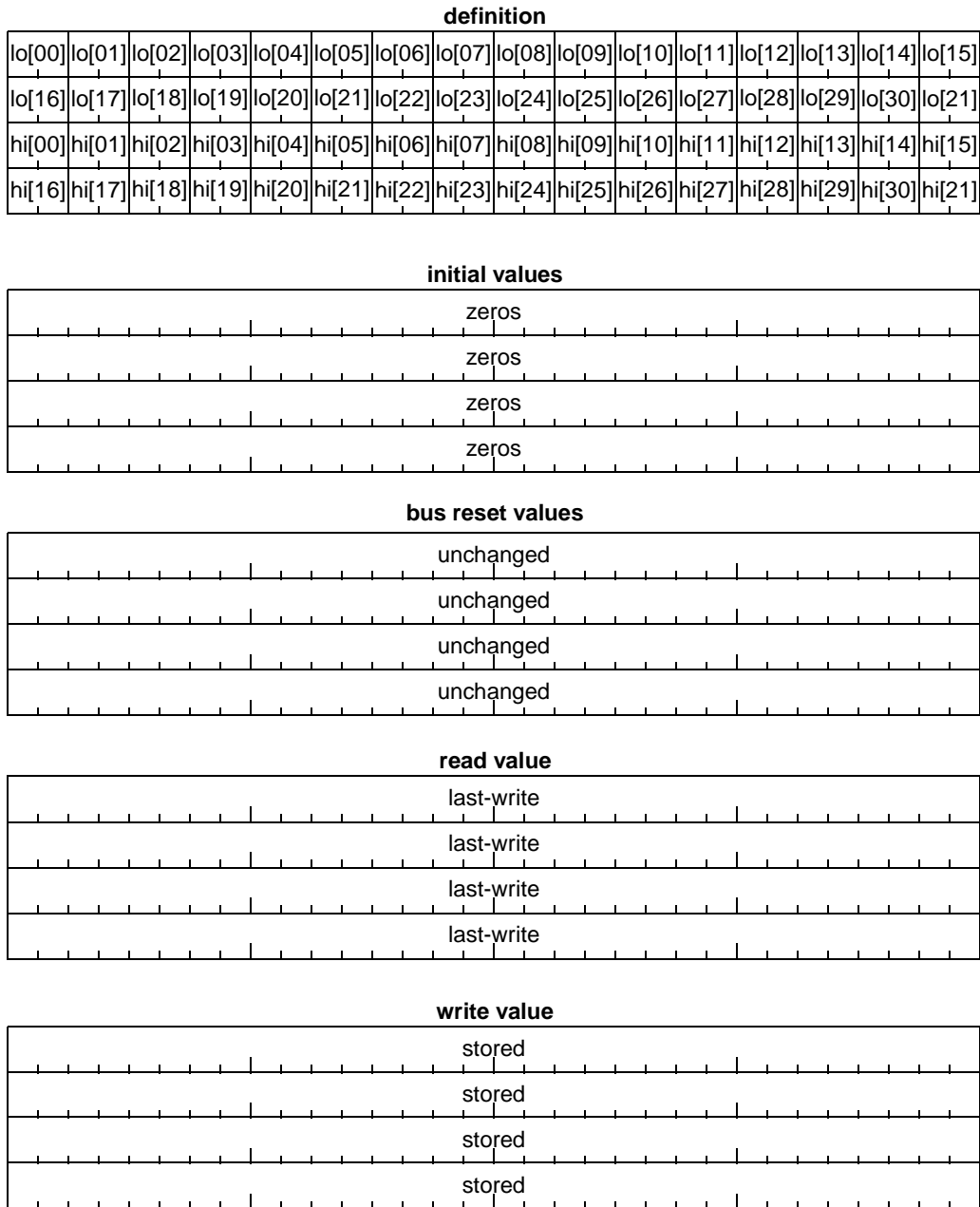


Figure B.1—Asynchronous transaction routing-tables

With the proper bit-mapped settings, the routing table can be initialized to support independent routing based on the more-significant half of the bus address, the less-significant half of the bus address, or inclusive/exclusive address ranges. Most of the regular topologies (hypercube or grids) can be also be supported, using either low-to-high or high-to-low dimensional routing (see B.3.2).

### B.1.2 ASYNC\_ROUTE tables

Bridges shall have a ASYNC\_ROUTE registers, located at offset TBD within each portal's CSR space. These registers provide access to that portal's routing table values, as illustrated in figure B.2.



**Figure B.2—ASYNC\_ROUTE register format**

A portal may provide additional registers to describe the cause or circumstances surrounding the error. The definition of such registers is beyond the scope of this standard.

## B.2 GASP packet routing

NOTE—This is an alternate proposal for routing of GASP packets, based on their *sourceId* address. It has the benefit of not having to setup channel number mappings and (more importantly) there is one fewer routing table to maintain.

Deadlock-free asynchronous stream routing decisions are made by checking the *sourceId* of passing-through packets. All asynchronous stream packets are accepted, but a *sourceId* check on the remote port causes some of the accepted packets to be discarded, as illustrated in figure B.3. If the directed-asynchronous routing is known to be deadlock free, the broadcast asynchronous-streams routing can be shown to be deadlock free.

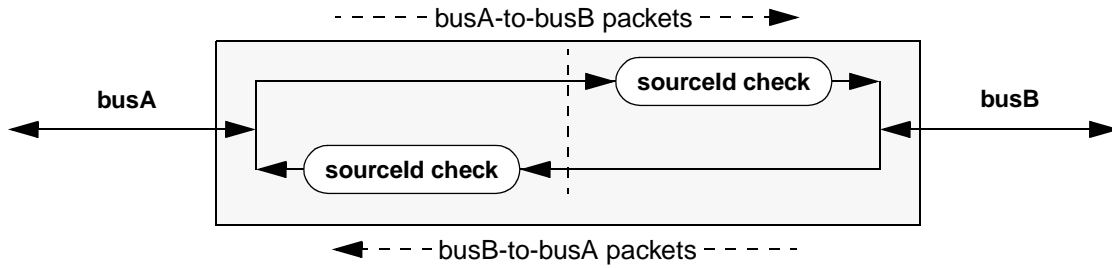


Figure B.3—Asynchronous stream routing

The *sourceId* checks are conceptually performed at the opposing portal, using the directed-routing tables in that portal. An asynchronous stream busA-to-busB packet with *sourceId*=*sId* is forwarded (rather than discarded) if routing tables on the busB port indicate that a request/response busB-to-busA packet with *destination\_ID*=*sId* would have been accepted.

An alternative, that can be easily supported when bridge ports are closely coupled, is to access the opposing run's response-check tables when the multicast packet arrives, accepting only those packets that will be retransmitted on the remote bus.

Routing the broadcast asynchronous streams in this fashion has several benefits:

- 1) Reduced costs. One set of routing tables is sufficient to route directed as well as broadcast packets.
- 2) Deadlock free. Any set of routing tables that ensures deadlock-free routing of directed request/response packets will also guarantee deadlock-free duplicate-free delivery of broadcast packets.

Although no additional cycle start routing tables are required: the request-*destination\_ID* routing tables must either be shared (which increases the worst-case access-bandwidth) or replicated (which increases the number of bridge registers). However, the additional hardware costs are small when compared to the software/hardware costs of maintaining and consistently updating a distinct set of address-routing tables.

When passing through a bridge, the GASP packets are expected to be placed in the request subaction queue, as opposed to the isochronous packet queue. This routing is intended to be consistent with the GASP packet's asynchronous arbitration properties.

### B.3 Larger N-portal bridges

Although initial bridge designs are expected to have only two portals, a general bridge architecture should allow more portals to be attached. To avoid accidentally constraining bridge architectures, several possible N-portal routing methodologies are considered.

The allocation of internal isochronous bandwidth is also more complex, and may depend on the details of the packet-router technology.

#### B.3.1 N-portal bridge distinctions

##### B.3.1.1 Simple N-portal bridges

A bridge may have more than two portals, but this additional flexibility increases routing-table complexity. During initialization, two base/bounds decisions (incoming and outgoing) must be made at each portal, as illustrated in figure B.4.

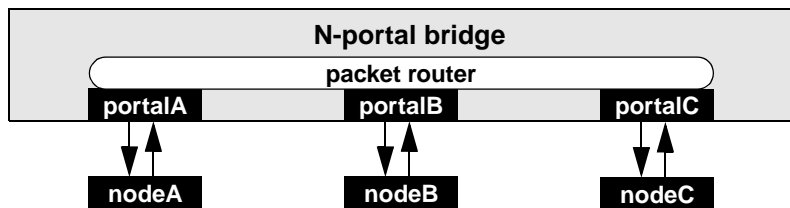


Figure B.4—N-portal bridge

The post-initialization routing tables may be different, but distinct or more-complex tables are needed to independently specify the bus-to-router and router-to-bus routing criteria

The 3-portal bridge is a specific instance of a generalized N-portal switch, where N is the number of portals on the bridge. A large bridge could, for example, have nearly 32K portals, where each portal connects to only one other node.

##### B.3.1.2 Generalized N-portal bridges

As was true for 2-portal bridges, other units may be collocated on a general-purpose N-portal bridge node, as illustrated in figure B.5.

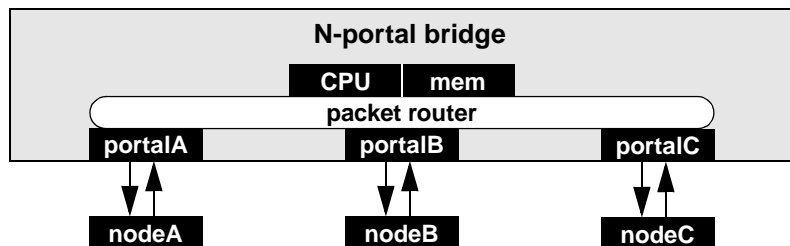


Figure B.5—N-portal multifunction bridge

### B.3.1.3 N-portal switches

Bridge portals are expected to have their own identity on the bus. However, switches only need to have one identity, so that nearly 64K attached nodes can be supported, as illustrated in figure B.6.

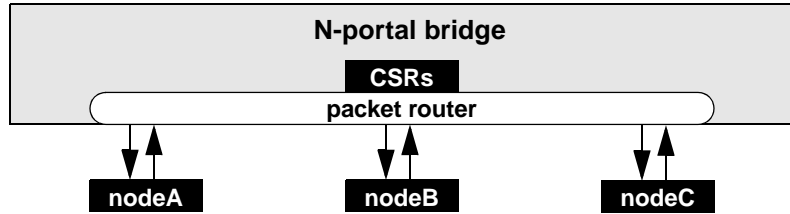


Figure B.6—N-portal switch identifiers

### B.3.2 Alternative routing-table structures

In an N-portal bridge, the most cost-effective routing methodology is highly dependent on the packet-router technology. As background, a variety of N-portal routing options are described. Although the N-portal routing decisions can be more complex, a simple base/bounds subset of the N-portal routing protocols is used during system initialization.

#### B.3.2.1 Destination-specified routing

Bridges with a small number of portals can have each portal broadcast its packet for processing by other attached portals. In such designs, each portal effectively provides the address-routing capabilities of a two-portal bridge. During initialization, two base/bounds decisions (incoming and outgoing) must be made at each portal, as illustrated in figure B.7. Two routing decisions are required: 1) bus packets are checked to determine which ones should enter the bridge and 2) router packets are checked to determine which ones should exit the bridge.

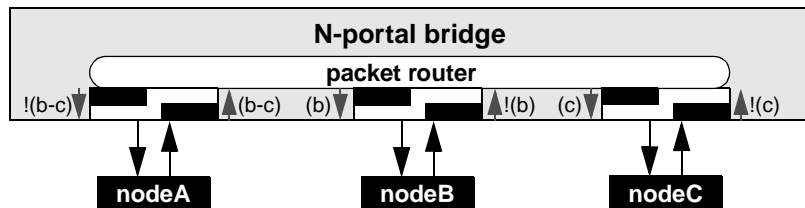


Figure B.7—Broadcast router checking

### B.3.2.2 Source-specified routing

To avoid inefficient broadcasts, bridged may be designed to route pre-labeled packets. When a packet from nodeA is accepted, the routing tables not only specify which addresses are accepted, they also specify the destination-portal address. The packet router could then route these packets through independent paths, based on their internal destination-label tag, as illustrated in figure B.8. In this illustration, the black box in each portal is the selective-acceptance routing tables; the diagonally-shaded box provides the destination-portal routing labels.

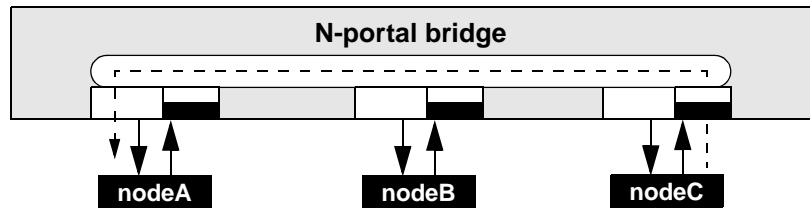


Figure B.8—Source-specified routing

Simple bridges could provide additional base/bounds registers and associate each range with a distinct portId; for an N portal bridge, N-2 base/bounds registers are required. For a 3-ported bridge, a single address threshold on each portal is sufficient to selectively-route packets towards one their two possible targets. On a 4-ported bridge, two address thresholds are sufficient to selectively route packets towards one of their three possible targets.

Because of their routing-table demands, general-purpose bridges are expected to have a limited number of portals (typically four or less). Larger bridges are expected to constrain their address-routing decision (radix-two switches, for example).

### B.3.2.3 Central routing

The labels that specify the portal-to-portal routing decisions could be located in a shared table, rather than replicated in each portal, as illustrated in figure B.9. For efficiency, the recent translations would be cached in each portal, so that source-specified routing could be used when the desired source-routing information is cached.

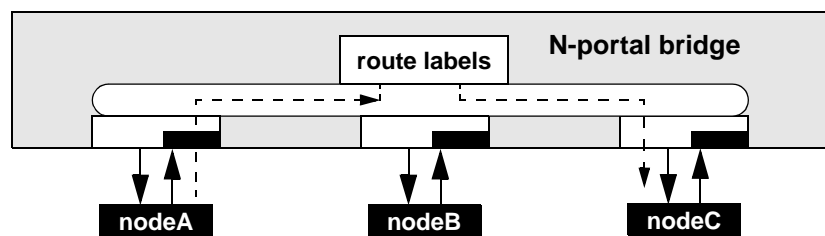


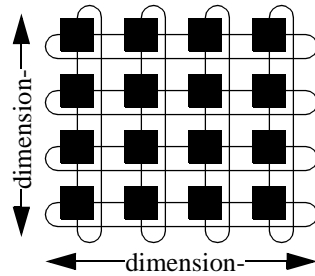
Figure B.9—Central routing tables

When the desired portal-cache entry is missing, the routing delay is increased by the need to fetch information from the central route-label hardware before the packet can be routed.

### B.3.2.4 Dimensional routing

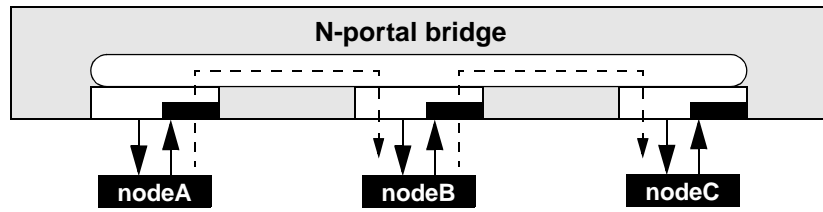
Many of the regular topologies (mesh and hypercube) are structured and have dimensions. To avoid deadlock, routing protocols can be dimensionally ordered: packets are routed in sequentially-increasing (or

sequentially decreasing, but not both) dimensional ordering. As a simple example, routing on a two-dimensional mesh is performed by routing in the vertical dimension (dimension-1) before routing in the horizontal dimension (dimension-2), as illustrated in figure B.10.



**Figure B.10—Simple dimensional routing**

For dimensionally-routed topologies, deadlock-free directed-transaction routing decisions are simple: a subaction sourced from a dimension N portal is routed to the dimension (N-1) portal, as illustrated in figure B.11.



**Figure B.11—Dimensional routing**

With dimensional routing, request and responses are expected to be routed in a complementary fashion: if the requests have low-to-high dimension routing, the returning responses would have a high-to-low dimension routing. This allows the responses to return through the path traced by their generating request.

## Annex C

(informative)

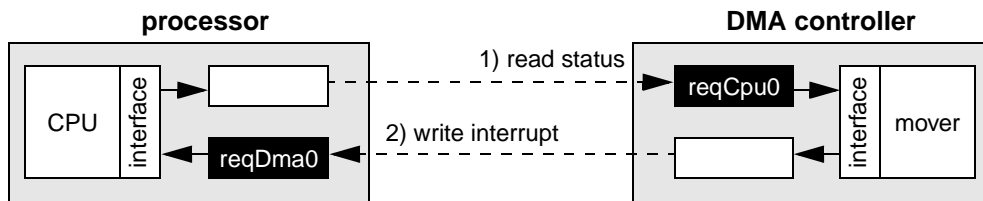
### Bridge considerations

#### C.1 Potential deadlocks

##### C.1.1 Potential queue-dependency deadlocks

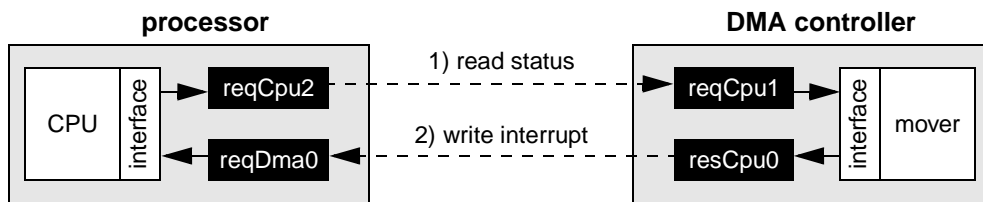
The simple hardware technique of placing request and response subactions in the same FIFO-ordered queue would create system deadlocks.

If requests and responses were to be placed in the same incoming and/or outgoing queues, deadlocks could occur. For example, consider a processor and DMA controller, as illustrated in figure C.1. A processor (not illustrated) attempts to read a DMA control register; this **reqCpu0** request is delivered-to and queued-in the DMA controller. A DMA controller (not illustrated) attempts to interrupt the processor; this **reqDma0** request is delivered-to and queued-in the processor.



**Figure C.1—Potential queue-dependency deadlocks, phase1**

Assume that the DMA controller processes its input request, generating a **resCpu0** response. Near the same time, another processor (not shown) attempts to read the DMA controller's status register; its **reqCpu1** request is delivered-to and queued-in the DMA controller. The processor now attempts to read another DMA control register; this **reqCpu2** request remains queued in the processor, as illustrated in figure C.2.



**Figure C.2—Potential queue-dependency deadlocks, phase2**

These two nodes are now deadlocked. On the processor, **reqDma0** cannot be processed because there is no queue space to hold an additional response. If **reqDma0** cannot be processed, **resCpu0** can never be sent. Similarly on the DMA controller, **reqCpu1** cannot be processed because there is no queue space to hold an additional response. If **reqCpu1** cannot be processed, **reqCpu2** can never be sent.

To avoid such deadlocks, Serial Bus nodes are expected to provide a pair of incoming queues (one for requests and one for responses) and a pair of outgoing queues (one for requests and one for responses).

### C.1.2 A bridge's potential queue-dependency deadlocks

Similarly, deadlock could occur if requests and responses were to be placed in the same bridge queues, even if the processor and DMA controller have pairs of request and response queues, as illustrated in figure C.3.

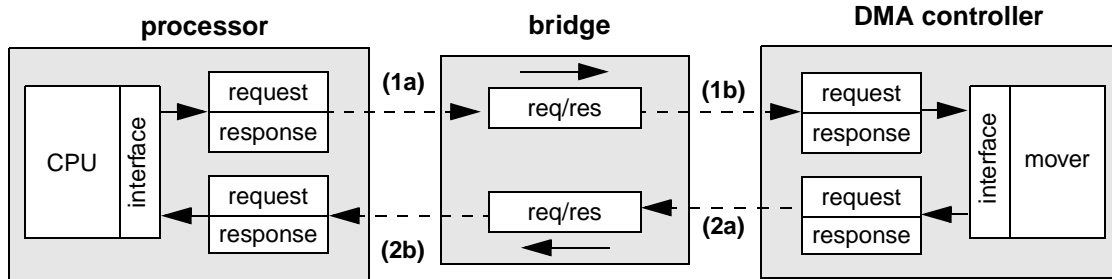


Figure C.3—Potential deadlocking bridge queues

The initiation of six transactions, by these or other nodes, is sufficient to deadlock the subaction queues, as illustrated in figure C.4.

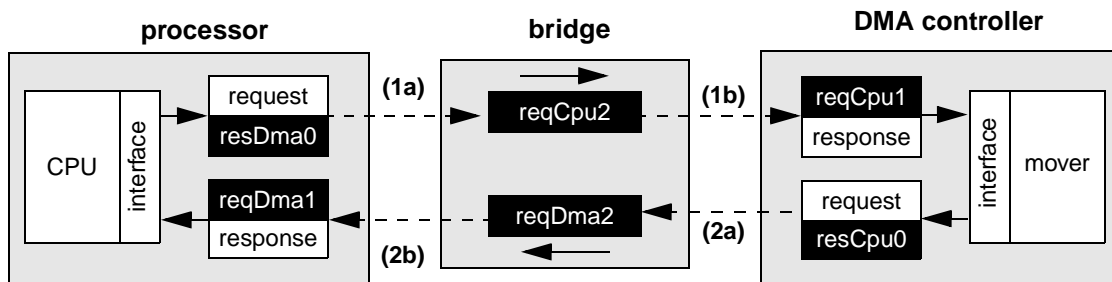


Figure C.4—Split-response read transaction

In this example, **reqCpu2** cannot be queued in the DMA controller until **reqCpu1** is processed by the DMA controller, **reqCpu1** cannot be processed by the DMA controller until **resCpu0** has been sent to the bridge, and **resCpu0** cannot be accepted by the bridge until **reqDma2** has been sent to the processor.

Similarly, **reqDma2** cannot be queued in the processor until **reqDma1** is processed by the processor, **reqDma1** cannot be processed by the processor until **resDma0** has been sent to the bridge, and **resCpu0** cannot be accepted by the bridge until **reqDma2** has been sent to the DMA controller.

To avoid such deadlocks, Serial Bus bridges are expected to provide a pair of right-to-left queues (one for requests and one for responses) and a pair of left-to-right queues (one for requests and one for responses).

## Annex D

(normative)

### Wormhole routing implications

Optional. This annex is only applicable to higher performance bridges that start retransmitting a packet on an adjacent bus B before the entire packet has been received on bus A. Lower performance store-and-forward bridges may ignore the contents of this annex.

#### D.1 Queued send-packet stomping

To effectively discard partially sent but erroneous send packets, a CRC at the end of the damaged packet is sometimes “stomped”. For example, if an error is introduced (1a) in the request-send packet, an error will be logged in the bridge and an `ack_data_error` will be returned, as illustrated in the left half of figure D.1.

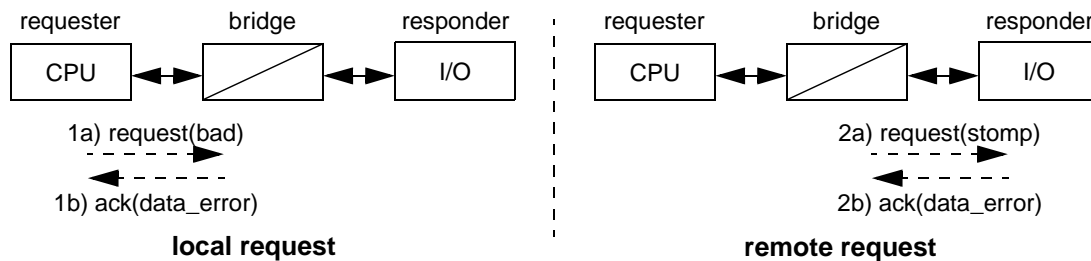


Figure D.1—Stomped Serial Bus packets

Note that the stomped request-send packet (2a) is generated by a high-performance pipelined agent, which begins to forward the send packet before checking its CRC. The remote responder could initiate its processing before the request-send packet’s CRC is verified, but must nullify any side-effects when the bad CRC is observed.

To better isolate errors, packet stomp values are exactly defined. Stomping involves setting the new CRC value to the exclusive-OR of *good* and *STOMP*, where *good* is what would have been the correct CRC value for the packet (as received) and *STOMP* value is defined in D.2. An error is logged if a packet’s CRC is different from the expected *good* and *stomped* values.

An error is expected to be logged the first time (and only the first time) that a packet is stomped, to assist in isolating the source of the error. For example, if an error is introduced on the portal2-to-portal3 link, the error will be logged in portal3, where the packet is first stomped.

#### D.2 CRC\_STOMP definitions

The following parameters are defined elsewhere in this standard:

The value of `CRC_STOMP` is defined to be `530F00FF16`; this is the value that is exclusive-OR’d with the correct CRC value when retransmitting the CRC of a known-to-be corrupted packet.



# **Annex E**

(normative)

## **Code listing**

```

// Arguments include the following:
// packetPtr - pointer to the isochronous packet
// newChannel - new channel number to be substituted
// diffCount - isochronous cycle in which packet was received
// Status values that are returned:
// PASS - non CIP-header packets pass through unprocessed
// MESS - CIP packet header format inconsistency, pass through unprocessed
// NONE - CIP header time not provided and therefore not adjusted
// DONE - CIP header time adjusted
ConvertInToOut(packetPtr, newChannel, phyId, diffCount)
{
    hPtr= (StdHeader *)packetPtr;
    switch (hPtr->tCode) {
    case OXA:
        // Normal and copy-protect isochronous
        break;
    default:
        assert(0);
    }
    /* Convert isochronous channel numbers */
    hPtr->channel= newChannel;
    /* Non CIP-header packets are not converted */
    if (hPtr->tag!=1)
        return(PASS);
    /* Packet should be large enough to contain the CIP header */
    if ((length= hPtr->data_length/4)<2);
        return(MESS);
    /* Check for CIP header flag-bit consistency */
    c0Ptr= (Cip0Header *) (packetPtr+2);
    c1Ptr= (Cip1Header *) (packetPtr+3);
    if (c0Ptr->eoch=1 || c0Ptr->form=1 || c1Ptr->eoch=1 || c1Ptr->form=1)
        return(MESS);
    /* The received sid field is transformed */
    c0Ptr->sid= phyId;
    /* Adjust the CIP header's cycle-count label */
    if (c1Ptr->fmt<=OXIF && c1Ptr->cycle_offset>=0XC00) {
        c1Ptr->cCount= (c1Ptr->cCount+diffCount)&0XF;
        return(DONE);
    }
    /* Compute&validate the source packet header location */
    spPtr= (SrcHeader *) (gpPtr+4*blockSize*((blockCount-c0Ptr->dbs)%blockCount));
    if (((int)spPtr >= packetPtr+(hPtr->data_length)/4)
        return(NONE);
    /* Adjust the source packet header's cycle-count label */
    spPtr->cycle_count= (c1Ptr->cycle_count+diffCount)%8000;
    return(DONE);
}

```

## Contact names

The following individuals have been monitoring the development of this standard, or have actively participated in the development of this standard. Their names are included to facilitate communications and thereby reduce the time needed to complete the standards development process. Listing of these individuals does not necessarily mean they concur with recent drafts or working group objectives.

***Email distribution list:***

geoffrey\_anderson@mail.sel.sony.com,  
akahane@wcs.sony.co.jp  
dbg@sunrise.scu.edu,  
davej@lsi.sel.sony.com,  
aaronl@lsi.sel.sony.com,  
Daniel.Meirsman@leu.ce.philips.com,  
sakmar\_gary@keithley.com,  
dicks@lsi.sel.sony.com,  
shima@usrl.sony.com  
scotts@lsi.sel.sony.com  
David.Wooten@COMPAQ.com

Geoffrey T. Anderson  
Sony Electronics  
1 Sony Drive  
Park Ridge, NJ 07656  
P: +1.201.930.6261  
F: +1.201.930.6397  
E: geoffrey\_anderson@mail.sel.sony.com

Masa Akahane  
E: akahane@wcs.sony.co.jp

David B. Gustavson  
SCIzzL Director  
1946 Fallen Leaf Lane  
Los Altos, CA 94024-7206  
P: 415 961-0305  
F: 415 961-3530  
E: dbg@sunrise.scu.edu

Jerry Hauck  
Silicon Group, Zayante, Inc.  
1580 Washington Blvd.  
Fremont, CA 94539  
F: 510-668-1457  
P: 510-668-1006  
E: jhauck@zayante.com

David V. James, PhD  
Sony Research Laboratories  
3300 Zanker Road, MS SJ 3D3  
San Jose, CA 95134  
P: +1.408.955.6295  
F: +1.408.955.6060  
H: +1.650.494.0926  
E: davej@lsi.sel.sony.com

Peter Johansson  
Congruent Software, Inc.  
98 Colorado Avenue  
Berkeley, CA 94707  
P: +1.510.527.3926  
F: +1.510.527.3856  
E: pjohansson@aol.com

Aaron Ludtke  
Sony Research Laboratories  
3300 Zanker Road, MS SJ 3D3  
San Jose, CA 95134  
P: +1.408.955.5226  
F: +1.408.955.6060  
E: aaronl@lsi.sel.sony.com

Daniel Meirsman  
Philips  
E: Daniel.Meirsman@leu.ce.philips.com

Farrell Ostler  
Philips Semiconductors, MS55  
9201 Pan American Fwy NE  
Albuquerque, NM 87113  
P: 505-822-7791  
F: 505-822-7836  
E: Farrell.Ostler@abq.sc.philips.com

Gary Sakmar  
Keithley Instruments, Inc.  
28775 Aurora Road  
Cleveland, OH 44139-1891  
P: 440 542-8016  
F: 440 248-6168  
E: sakmar\_gary@keithley.com

Dick Scheel—p1394.1 Chair  
Sony Electronics  
2350 Mission College Blvd, Ste. 982  
Santa Clara, CA 95054  
P: +1.408. 982.5834  
F: +1.408. 982.5899  
C: +1.408.307.1696  
E: dicks@lsi.sel.sony.com

Hisato Shima  
E: shima@usrl.sony.com

Scott Smyers  
Sony Research Laboratories  
3300 Zanker Road, MS SJ 3D3  
San Jose, CA 95134  
P: +1.408.955.4573  
F: +1.408.955.6060  
E: scotts@lsi.sel.sony.com

David Wooten  
P: +1.281.518.7231  
E: David.Wooten@COMPAQ.com