

Recognized as an American National Standard (ANSI)

IEEE Std 1212.1-1993(2001)

IEEE Standard for Communicating Among Processors and Peripherals Using Shared Memory (Direct Memory Access—DMA)

Sponsor

**Microprocessor and Microcomputer Standards Committee
of the
IEEE Computer Society**

Approved June 17, 1993
Reaffirmed December 6, 2001

IEEE Standards Board

Approved December 2, 1993
Reaffirmed May 2, 2002

American National Standards Institute

Abstract: Primitive yet high-performance means are defined for passing messages across the bus between the Processor and some form of intelligence in the I/O Unit's Function. This message-passing scheme makes minimal demands on the instruction set and hardware required. In addition, several simple conventions are defined for the structure of the messages passed. The intent is to provide a standard architectural framework that supports the detailed definition of application-dependent I/O Unit and Function interface standards. The algorithms and definitions themselves are useful in the design of integrated circuits for I/O.

Keywords: direct memory access, DMA, memory, I/O, processor

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1994 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 1994. Printed in the United States of America

ISBN 1-55937-352-0

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE standards documents may involve the use of patented technology. Their approval by the Institute of Electrical and Electronics Engineers, Inc. does not mean that using such technology for the purpose of conforming to such standards is authorized by the patent owner. It is the obligation of the user of such technology to obtain all necessary permissions.

Introduction

[This introduction is not a part of IEEE Std 1212.1-1993, IEEE Standard for Communicating Among Processors and Peripherals Using Shared Memory (Direct Memory Access—DMA).]

This document describes a DMA Framework that is an optional addition to the IEEE Std 1212-1991, IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses. The CSR Architecture defines a register set to be used for control, configuration, identification, and diagnostics between nodes on a system bus interconnect. The CSR Architecture and DMA Framework documents may be cross-referenced by other bus specifications, such as Futurebus+ (the IEEE 896 series) or Serial Bus (IEEE P1394), other bus standards developed in the future, as well as other standards in the IEEE 1212 series itself. The goal of the IEEE Std 1212 CSR Architecture is to provide a scalable, extensible bus-independent framework for interoperability among nodes capable of communicating across a system interconnect. The goal of this document is to specify the recommended architectures and standard building blocks with which standard I/O Function interfaces can be built (e.g., LAN and disc interfaces).

This document evolved from rough draft documents originally developed within the P1212 Working Group. Work was later moved to a separate working group in order to expedite completion of the IEEE Std 1212-1991 CSR Architecture. It is anticipated that additional documents will be added to the IEEE 1212 series in the future (for example, for standard system boot, specific standard I/O Unit interfaces, etc.) and that standards in other series will also be based on this work.

It is not necessary to read IEEE Std 1212-1991 in order to obtain an overall understanding of the models in this document. Also, many aspects of this framework are expected to be useful in systems that are not compatible with IEEE Std 1212-1991.

The concepts in this document are based primarily on the work of Jon Crowell, Steve Dean, Sam Duncan, Frank Fiduccia, Jim Leahy, Richard Napolitano, Dan O'Connor, Chet Pawlowski, and Mike Wenzel.

The Chair and Editor gratefully acknowledge the many working group members who contributed their time and ideas to this effort.

The following is a list of those who were members of the P1212.1 Working Group while the draft and final specification were compiled:

Sam Duncan, *Chair*
Mike Wenzel (*Editor*)

Barbara Aichinger
Harrison Beasley
Kyle Black
Paul L. Borrill
Jay Cantrell
Mike Chastain
Jon Crowell
W. P. Evertz
Frank Fiduccia
Claes-Goren Gustavsson

Dave Gustavson
Emil Hahn
Craig Hansen
Mark Hassel
Ed Jacques
David V. James
Anatol Kaganovich
Jim Leahy
Thanos Mentzelopoulos
Richard Napolitano

Dan O'Connor
Calvin Olsen
Chet Pawlowski
Uri Rahamim
Don Senzig
Tom Tate
Michael Teener
Martin Whittaker
Mark Woodbury

The following persons were on the balloting committee:

Gyenes Alexandre
Richard P. Ames
Bjorn Bakka
Harrison A. Beasley

Janos Biri
John Black
William P. Blase
John C. Brightwell

Haakon Bugge
Greg Buzzard
David Cohen
Graham Connolly

Jonathan C. Crowell
 Samuel Duncan
 Sourav Dutta
 R. D. Edwards
 Wilhelm P. Evertz
 Wayne Fischer
 Gordon Force
 David B. Gustavson
 Sandor V. Halasz
 Mark Hassel
 Zoltan R. Hunor
 Edgar Jacques
 David V. James
 Kenneth Jansen
 Sverre Johansen
 Ernst H. Kristiansen
 Thomas M. Kurihara
 Ralph Lachenmaier

G. Laws
 Min-Suk Lee
 Stephen Mabbs
 Tulin Mangir
 Arthur McNealus
 S. Millendorf
 Svein Moholt
 D. Mounday
 J. D. Nicoud
 Daniel C. O'Connor
 Calvin Olsen
 Michael Orlovsky
 Donald Pavlovich
 Robert K. Rouse
 Norman Schneidewind
 D. Senzig
 K. Shore
 Michael R. Sitzer

David Stevenson
 Robert Stewart
 Dongzhuang Su
 Michael Teener
 Robert Tripi
 Yoshiaki Wakimura
 Michael Wenzel
 Hans A. Wiggers
 Dwight Wilcox
 Mark Williams
 Andrew Wilson
 Joel Witt
 David L. Wright
 Oren Yuen
 Janusz Zalewski
 George R. Zerdian

When the IEEE Standards Board approved this standard on June 17, 1993, it had the following membership:

Wallace S. Read, *Chair*
Donald C. Loughry, *Vice Chair*
Andrew G. Salem, *Secretary*

Gilles A. Baril
 José A. Berrios de la Paz
 Clyde R. Camp
 Donald C. Fleckenstein
 Jay Forster*
 David F. Franklin
 Ramiro Garcia
 Donald N. Heirman

Jim Isaak
 Ben C. Johnson
 Walter J. Karplus
 Lorraine C. Kevra
 E. G. "Al" Kiener
 Ivor N. Knight
 Joseph L. Koepfinger*
 D. N. "Jim" Logothetis

Don T. Michael*
 Marco W. Migliaro
 L. John Rankine
 Arthur K. Reilly
 Ronald H. Reimer
 Gary S. Robinson
 Leonard L. Tripp
 Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
 James Beall

Richard B. Engelman
 David E. Soffrin

Stanley I. Warshaw

Paula M. Kelty
IEEE Standards Project Editor

CLAUSE	PAGE
1. Overview.....	1
1.1 Scope of DMA recommendations.....	1
1.2 Objectives.....	2
1.3 Overview of DMA	3
2. References.....	11
3. Document structure and notation	11
3.1 Document structure and conformance requirements	11
3.2 Conformance levels	12
3.3 Technical glossary.....	12
3.4 Data structure notation.....	16
3.5 Node description (capitalization)	19
3.6 Notation used for algorithms.....	19
4. Background concepts	21
4.1 Miscellaneous components	21
4.2 Circular Queue	26
4.3 Dispatch List	28
4.4 Shareable List.....	30
4.5 Mailbox	34
4.6 Hybrid DMA schemes	37
4.7 Device-to-Device DMA.....	40
5. Specification of Circular Queues as a DMA interface	43
5.1 Overall description of Circular Queue DMA applied to I/O	43
5.2 Circular Queue data structures.....	45
5.3 Specific services Circular Queues require from underlying bus system	46
5.4 Unit initialization steps for Circular Queues.....	46
5.5 Circular Queue enqueue operations	46
5.6 Circular Queue dequeue operations	47
6. Specification of Dispatch Lists as a DMA interface.....	49
6.1 Overall description of Dispatch List DMA applied to I/O.....	49
6.2 Dispatch List data structures	50
6.3 Specific services Dispatch Lists require from underlying bus system.....	51
6.4 Unit initialization steps for Dispatch lists	52
6.5 Dispatch list enqueue operation	52
6.6 Dispatch List dequeue operations	53
7. Specification of Shareable Lists as a DMA interface	54
7.1 Overall description of Shareable List DMA applied to I/O	54
7.2 Shareable List data structures	55
7.3 Specific services Shareable Lists require from underlying bus system	56
7.4 Unit initialization steps for Shareable Lists	57
7.5 Shareable List enqueue operations.....	57

CLAUSE	PAGE
7.6 Shareable List dequeue operations.....	59
8. Specification of Mailboxes as a DMA interface.....	61
8.1 Overall description of Mailbox DMA applied to I/O	61
8.2 Mailbox data structures	62
8.3 Specific services Mailbox DMA requires from underlying bus system	63
8.4 Unit initialization steps for Mailbox DMA.....	63
8.5 Mailbox enqueue operations	63
8.6 Mailbox dequeue operations	64
9. Specification of Device-to-Device DMA	65
9.1 Device-to-Device DMA data structures.....	65
9.2 Specific services required from the underlying bus system.....	67
9.3 Unit initialization steps for Device-to-Device DMA	68
9.4 Producer I/O Unit enqueue operations for Device-to-Device DMA	68
9.5 Consumer I/O Unit operations for Device-to-Device DMA.....	69
9.6 Producer unit operations for buffer return	70
9.7 Producer unit operations for terminating Device-to-Device DMA	71
10. Specifications and conventions for DMA message structure	72
10.1 Overview of message structure	72
10.2 DMA message specifications	72
10.3 Guidelines for DMA message conventions	77
11. Details: Specifications applying to all models	84
11.1 Services assumed from the underlying bus system.....	85
11.2 DMA Controller functions	90
11.3 Initialization, powerfail recovery, and reset.....	93
11.4 Interrupt mechanisms	100
11.5 Other specifications applying to all models.....	100
Annex A (Informative) Example applications	102
Annex B (Informative) Caching issues.....	111
Annex C (Informative) Notes on I/O Unit Design.....	114
Annex D (Informative) Coarse performance comparisons and tradeoffs	116
Annex E (Informative) Description of former Circular Queue options.....	125
Annex F (Informative) Bibliography	128

IEEE Standard for Communicating Among Processors and Peripherals Using Shared Memory (Direct Memory Access—DMA)

1. Overview

1.1 Scope of DMA recommendations

This document describes a DMA Framework that provides recommended architectures for high-performance interfaces between Functions¹ and System Memory, connected by buses that are generally compatible with the IEEE Std 1212-1991 CSR Architecture,² such as Futurebus+ [B1] , [B2] and Serial Bus [B4] .³ These buses are characterized by high bandwidth, but have significant delays compared to Processor access to cache, particularly when used to interconnect multiple buses or large numbers of mostly independent units. The design models defined by this DMA Framework assume that transaction latencies could be large. Four models for Processor to I/O Function communication and one model for I/O Function to I/O Function communication are described. Other designs may be appropriate for single-board systems, where the I/O related Function has immediate access to System Memory and Processor resources.

Other DMA architectures are possible in systems conforming to IEEE Std 1212-1991. It must be emphasized that the objective is *not* to arrive at a single DMA model that will be welded forever to IEEE Std 1212-1991. DMA specifications should be optional to allow for newer and better schemes, or ones more suited to a given application. Yet a point of departure is needed that will guide the initial development of common interfaces.

The approach of this document is to define very primitive yet high-performance means of passing messages across the bus, between the Processor and some form of intelligence in the I/O Unit's Function (e.g., an I/O card processor or intelligent frontplane chip).⁴ This message-passing scheme should make minimal demands on the instruction set and hardware required. In addition, several simple conventions are defined for the structure of the messages passed to do

¹Refer to 3.3 for definitions of terms.

²Information on references can be found in clause 2.

³The numbers in brackets preceded by the letter B correspond to those of the bibliography in annex F.

⁴All the models in this document assume relatively intelligent I/O Units that can provide a “do this”/“OK, I did it”-style message interface. This minimizes the interrupts and Processor stalls that would otherwise be necessary to read Unit CSRs. The working group was unable to define general, device-independent DMA models for non-intelligent units.

common operations in a common way. The most important convention is the vector structure used for representing buffer segments of user data. These message-passing and structure definitions form the building blocks for specific interfaces; for example, LAN, disk, and printer interfaces.

This document is *not* a complete interface standard. It is a standard architectural framework that supports the detailed definition of application-dependent I/O Unit and Function interface standards, but standard general-purpose bus interfaces and DMA-related hardware, to support one or more of these standards, can be designed that use the primitive definitions of this document.

This framework is recommended to guide system designs. The intention is to maximize inter-vendor compatibility and to maximize the possibility that general-purpose integrated circuits will be available for a wide range of applications.

1.2 Objectives

The DMA Framework was designed to meet the following objectives:

- 1) *Configuration flexibility*—The DMA Framework should be capable of being implemented on a wide range of system configurations, including the following:
 - a) *Wide range of peripherals*—Including disk, tape, and communications devices, as well as system-internal memory-to-memory transfers. These peripherals should be directly supportable, or should be supportable indirectly using shared communication links (like SCSI or IPI). A peripheral may also be a processor, since the DMA model is useful for processor-to-processor communications.
 - b) *Coherent and noncoherent systems*—Allow I/O Units to efficiently support cache-coherent, noncoherent, or partially coherent systems (see annex B).
 - c) *Multiple implementation technologies*—Support unit implementations based on powerful processors, I/O coprocessors, low-end microprocessors, or custom ICs.⁵
 - d) *Processor/memory environment*—Be compatible with the CSRs and transaction set defined by the IEEE Std 1212-1991 CSR Architecture. The cost of specialized hardware that is provided exclusively to support the DMA Framework should be minimized. Optional transactions should not be used, since these are not supported on all bus standards.⁶
 - e) *Physical addressing*—Be based on physical addresses rather than virtual. The support of virtual addressing would be more expensive and is highly vendor-dependent.
 - f) *Configuration size*—Support designs optimized for simple single-bus uniprocessor systems as well as for multiple-bus systems with large numbers of multiprocessors.
 - g) *Reliability*—Maximize system reliability for a wide range of configurations where I/O nodes may be independently powered or reset.
 - h) *Backward compatibility*—Avoid requiring modification of existing processors, memory controllers, and bus bridges on system buses that are bridged to IEEE Std 1212-1991 compatible buses.
- 2) *Architecture flexibility*—The DMA Framework should efficiently support a wide range of I/O driver software environments, whose requirements include but are not limited to the following:
 - a) *Multiple I/O transactions*—Support the queuing of multiple I/O transactions on a single DMA channel. Once they are removed from the queue, the processing order for these I/O transactions may differ from their enqueueing order (actions may be completed out of order).
 - b) *Interrupts*—Support synchronous and asynchronous interrupts. Synchronous interrupts are generated at the completion of a pre-specified data transfer. Events not associated with the completion of active DMA transfers generate asynchronous interrupts.

⁵Except for processor-interrupt registers, avoid specifying CSR read or write characteristics that would require custom hardware. Treat each CSR as though it could be read or written by a microprocessor in the Unit. This would include avoiding bits that don't set when written, reads that produce side-effects, writes that cause instantaneous results, etc.

⁶Operations should especially be avoided that retain bus tenure for multiple bus transactions in order to obtain an atomic result (e.g., read-modify-write), since these are not well supported by many existing bus standards. Similarly, minimize the use of noncoherent spin-locks. These operations waste bus cycles, waste processing time, distort system priorities, and frequently result in contention for a lock. Note that this discussion does not apply to swap or similar transactions that do not tie up the bus and that obtain an atomic result by combining operations inside the destination unit (e.g., within a memory controller).

- c) *Contiguous memory*—Avoid the use of physically accessed data structures that are larger than one page (4 Kbyte typical) in size. On a virtual-memory-based system, such contiguous physically addressed data structures are hard to allocate.
- 3) *Performance*—The DMA Framework should equal or surpass the performance provided by alternative DMA architectures on today's systems. The following design strategies are used to achieve this objective:
 - a) *CSR Reads*—Avoid requiring the Processor to read CSRs on an I/O Unit for normal operation. Such reads are expected to be slow, and the Processor is expected to stall until the read has completed.
 - b) *CSR Writes*—Require the Processor to write at most one Unit CSR for each I/O transaction. Note that the latency of CSR writes can often be hidden by buffering the write (the Processor continues before the write completes at the Unit).
 - c) *Transfer size*—Efficiently support small (< 20 bytes) as well as large (≥ 4 Kbyte) transfers.
 - d) *Transaction counts*—Minimize the number of bus transactions required to implement a DMA transfer. Each transaction that is not part of a contiguous data transfer has some setup overhead.
 - e) *Interrupt counts*—Interrupt the Processor at most once per I/O transaction. When multiple I/O transactions are completed nearly simultaneously, these interrupts should be combinable into one.
 - f) *Interrupt efficiency*—Require the DMA Unit to provide sufficient information for the Processor to quickly dispatch the proper interrupt-service routine. This should be possible even when large numbers of Functions share the same processor interrupt bit.
 - g) *Overhead*—Minimize data moved through the bus that is of no interest to the consumer or that is only an artifact of the DMA model itself, and this data should provide efficient means for the producer to retrieve state information related to the I/O transaction.
 - h) *Optimization*—Optimize for typical command and data sizes, and provide expansion mechanisms to cover the worst case.

1.3 Overview of DMA

1.3.1 Bus topologies considered

The primary emphasis of this document is on the communication between Processor units and I/O Units (although these concepts also may be used for Processor-to-Processor communication). This clause will discuss the assumed environment for this communication, some problems considered, and the paradigm for communication.

Figure 1.1 shows the topologies considered. These include both single-bus and multiple-bus systems, the latter having one or more levels of bus bridges. Processors, I/O Units, and bridges may be cache-coherent, noncoherent, or mixed. Mixed coherence occurs when the Processors are cache-coherent but the I/O Units and possibly the bridges are not (please refer to annex B for more details on mixed coherence). I/O Units are normally expected to outnumber Processors, possibly by as many as two orders of magnitude in large systems. All I/O Units of interest have read/write access to System Memory. Therefore, DMA hardware is normally part of the I/O Unit itself, rather than a separate bus entity. Generally, this read/write access is characterized by high bandwidth but significant delay—much larger delay than when accessing first-level cache or other intra-nodal RAM.

For performance reasons, Processors are expected to reside on the same bus as System Memory and are expected to share the same power source. On large, multiple-bus system topologies, I/O Units may reside on remote buses that are separately powered. This leads to the primary system failure mode. DMA protocols should be robust, so that a bus that has lost power does not affect the continued operation of the remaining buses. For example, if an I/O Unit on a remote bus loses power, data structures and locks should be left in a non-corrupt state so that the rest of the system can continue to function.

1.3.2 I/O transaction flow and use of memory

The communication of interest here is symmetrical at the most primitive message-queuing level, and mostly asymmetrical at the higher memory-management and message-semantic levels. For common I/O transactions, the Processor is the initiator and the I/O Unit is the responder. The Processor provides the related buffer space and

generally functions as a master over the I/O Unit. The initiation/completion message handshake pair is expected to be a very common paradigm for DMA communication, independent of Processor-to-I/O Unit relationships, in that it is also expected to be useful for Processor-to-Processor or I/O Unit-to-I/O Unit communication. A typical I/O transaction illustrating all of these concepts may proceed as follows (see figure 1.2):

- 1) The Processor sets aside data buffers in System Memory, and ensures that this data is accessible by I/O Units. If I/O transactions are not performed coherently with the Processor's cache (by the I/O Unit or a bus bridge), this may require explicit flushing of Processor cache lines.
- 2) The Initiator (Processor) puts a transaction-initiation message in the initiation (outbound) queue corresponding to the Responder (I/O Unit). This may involve placing the message in an initiation queue in System Memory or writing it directly to Unit registers.
- 3) The Initiator triggers the Responder to service the initiation queue.
- 4) The Responder consumes the message from the initiation queue, parses explicit commands in the initiation message (if any), and schedules the message-related event for Function processing.
- 5) During event processing, the Responder block-copies data between System Memory and a remote device or memory area.
- 6) When the data in System Memory is stable, the Responder (I/O Unit) puts a transaction-completion message in the completion (inbound) queue for the Initiator (Processor).
- 7) The Responder triggers the Initiator to service the completion queue.
- 8) The Initiator consumes the transaction-completion message.
- 9) The Initiator (Processor) delivers read data and status to the user's application and reclaims the data buffer space.

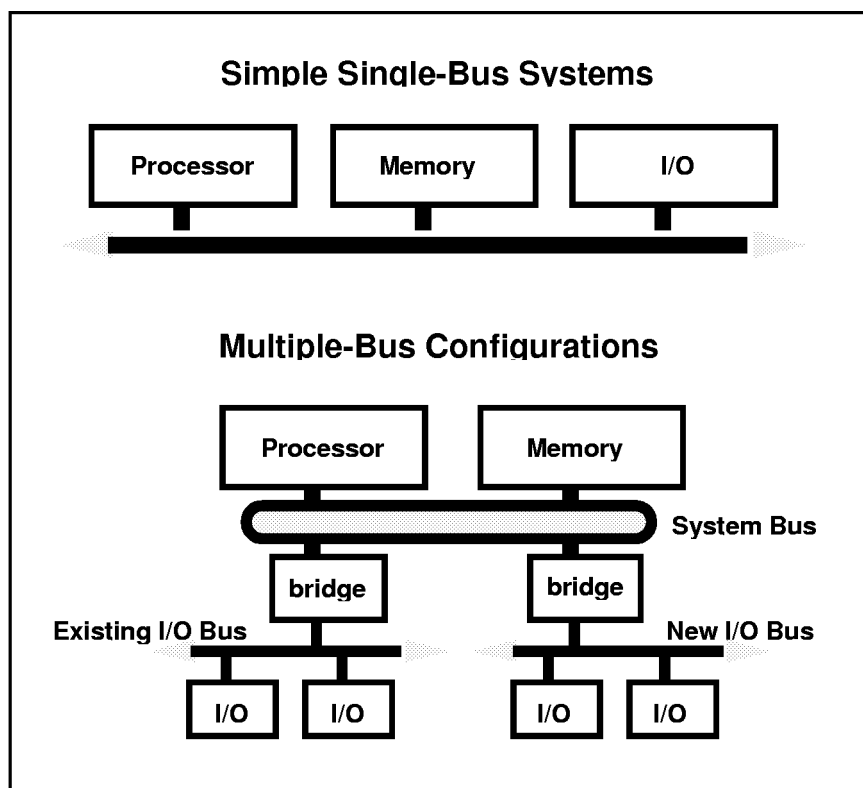


Figure 1.1—Representative bus topologies

The DMA models in this document will concentrate primarily on the operation of the queues themselves, since synchronization of operations between the producer(s) and consumer is the primary problem being addressed. The

queue items represent “event” messages that pass through the bus, either containing data and commands directly or pointing to buffers in System Memory. Recommended message contents and conventions (like the I/O transaction-initiation/completion paradigm) are optional. For example, for a disk interface, an outbound queue item could represent the I/O transaction initiation for a disk seek, and an inbound queue item could represent I/O transaction completion. Queue items also may be used for one-way Processor-to-Unit commands or asynchronous Unit-to-Processor status indications, such as a signal that a new disk has come on-line. DMA queues may be used in similar initiation/completion or one-way paradigms for peer-to-peer communication—Processor-to-Processor or device-to-device.

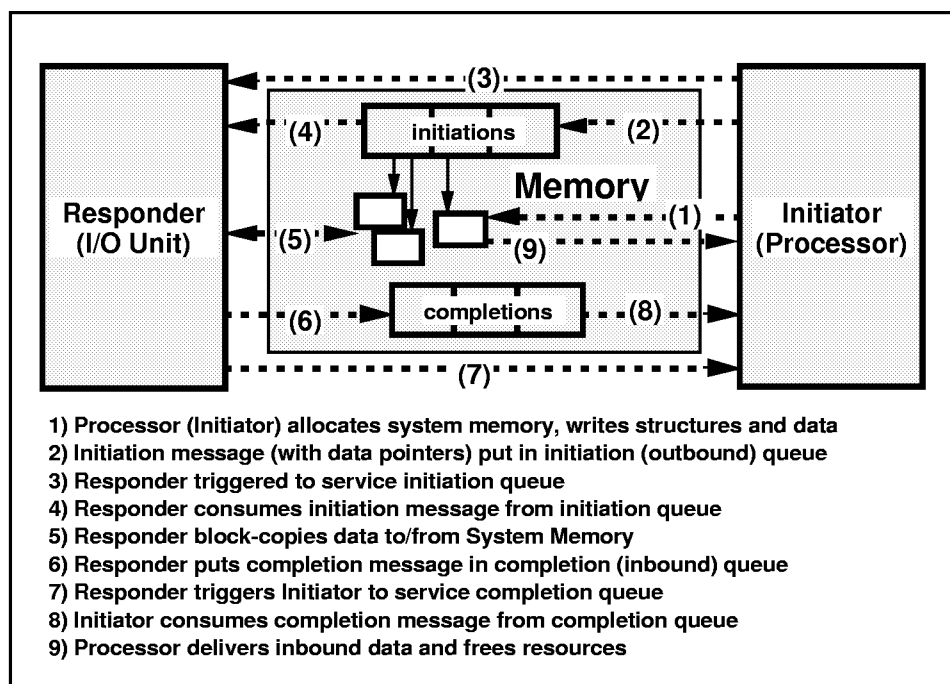


Figure 1.2—DMA execution model

Note that the messages and signals may not be related one-to-one. Several buffers may be passed at once in Step 2 or several completions signalled at once in Step 6. Also, for some applications, a one-way message may be delivered via the outbound or inbound queues with only a simple signal of consumption. Furthermore, some DMA schemes can combine steps.

Since I/O Units have access to System Memory, the communication paradigm employed is a hybrid involving both message-passing and shared memory. Buffers and parameter block structures involving bulky information or needing long-term storage are located in System Memory. “Long-term” is used in this document to mean either tens of milliseconds (e.g., during a disk latency) or seconds (e.g., for a data link connection that is blocked by flow control.) The transaction initiation and completion events themselves are represented by messages. These event messages point to the shared memory areas. They also contain a sufficient amount of information for the receiving unit to know the operations involved and to find the resources required to process the event.

Event messages between the Processor and I/O Unit are intended to be queued for only a very short time until the receiving unit can service a related interrupt or poll for message events. Therefore, primary_messages are normally used to pass events across the bus. Function-dependent software and firmware in both the Processor and the I/O Unit's Function provides long-term storage of event-related state information. Queuing delays for primary_messages should be short. Therefore, there should be little need for priority queuing of the primary_messages themselves. Handling of event priorities should be provided by Unit-dependent functions that decide how each new event should be handled

with respect to other events and activities.⁷ So priority queuing of the event will be handled after the event message is received (see the discussion of priority handling below).

Note that the discussion in this clause concentrates primarily on Processor-I/O Unit interactions with some degree of state and semantic content. The queue structures in this document may be used for long-term storage when simple FIFO processing is involved: e.g., for a stateless, datagram I/O Unit Function or for a Processor to supply pools of empty memory blocks to an I/O Unit.

1.3.3 Node and queue topology

In the following discussion, recall that the IEEE Std 1212-1991 CSR Architecture defines a *node* as having a 4 Kbyte (minimum) initial address space containing a set of defined control and status registers. The node contains the standard registers involved in self-identification, system power-on, fatal error handling, etc.—the basic registers involved in system hygiene that are not normally involved in the main performance path. A *unit* is a subcomponent of a node that provides processing, memory, or I/O functionality and a register interface for I/O driver software.

In this document, we define an *I/O function* as being a logically separate component of a unit that may or may not have external registers and that operates mostly independently (e.g., a LAN or disk frontplane controller). Since the DMA schemes involve the use of registers, DMA queues are owned by units. Of interest here is that one or more functions may be reached via a given DMA queue.

Subclause 1.3.2 described the simplest node and queue topology. I/O nodes will often contain more than one Function. This subclause considers some of the alternatives for Function-to-Node relationships and how they can affect the layout of DMA queues. The selection of the optimal system configuration depends on the physical packaging, system performance, system cost, and application. Although the DMA Framework can support any of these multi-function system topologies, the selection of the appropriate topology is beyond the scope of the DMA Framework.

It is important to understand that message queues link *physical* entities, like Processor and I/O Unit processors or intelligent frontplane controllers. The DMA hardware for a unit will normally only support a small number of physically separate queues (e.g., 2 to 8). For this reason, all messages between a pair of physical entities will normally be multiplexed onto a small number of queues—often a single pair: one for each consuming entity. As far as the message-passing mechanisms are concerned, there is no relationship among messages delivered to a given queue. Unit and Function specifications superimpose multiplexing relationships on the message contents, when necessary.

1.3.3.1 One function per node

The DMA queue applications are described in terms of unit architectures, where one I/O Unit and one Processor unit are associated with each pair of inbound and outbound queues. In the simplest conceptual implementations, a single Function would be contained within a Unit and a single Unit would be contained within a node.

One pair of inbound-outbound queues would be associated with each Unit (and in this illustration, each node) and the Function would be accessed through the queue pair associated with the containing unit, as illustrated in figure 1.3 (which does not show the similar node-unit-function composition for the node containing the system Processor). Since all DMA schemes require the use of CSRs, the queues must be managed by Unit hardware, not by the functions directly. The figure illustrates this by the way the boxes are nested.

⁷The handling of event priorities and commands (e.g., device or connection resets) can vary widely among applications. The idea here is to isolate the primitive, low-level, message-passing mechanisms from complex application dependencies. See the discussion in annex C.

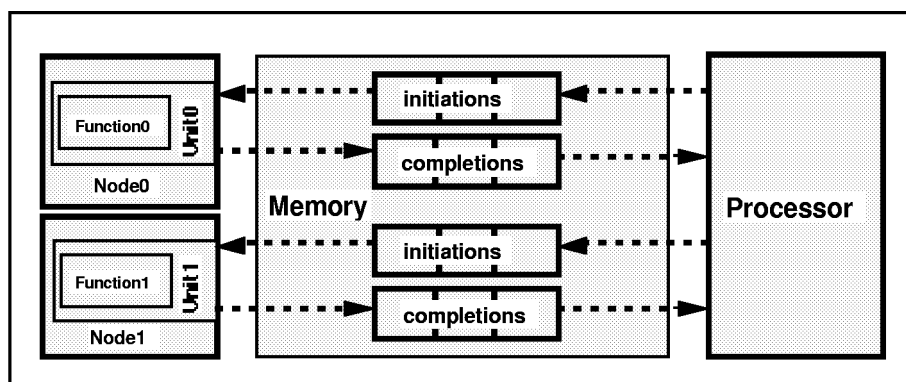


Figure 1.3—One function per node

1.3.3.2 One function per unit

Multiple Functions could be placed on one node by defining distinct Units, with one Unit register set for each of the Functions, as illustrated in figure 1.4 (which, again, does not show the composition of the system Processor's node). Although this is a natural extension of the one Function per node model, it may be more cost-effective to share the DMA queue pair among multiple Units, as described in 1.3.3.3.

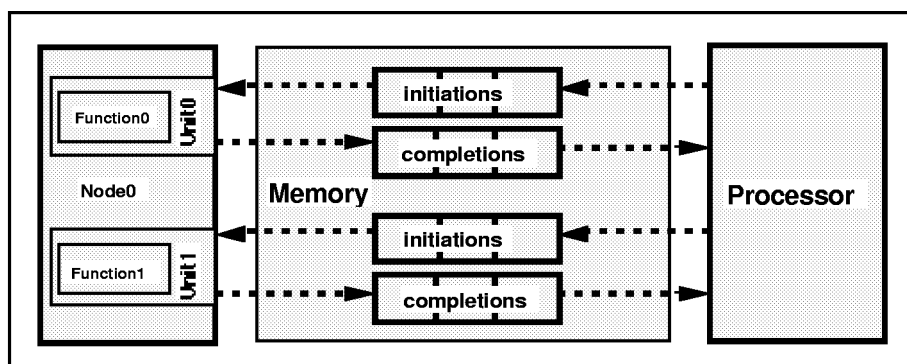


Figure 1.4—Multiple units per node

1.3.3.3 Multiple functions per unit

Multiple functions on one node could share a single register set and DMA queue pair. In this case, the multiple functions share a common resource (i.e., the register set and queues) and are therefore part of a single unit. The Unit hardware uses a message field, like the `channel_number` to demultiplex outbound messages and deliver them to the related function, as illustrated in figure 1.5.

Figure 1.5 also illustrates the composition of the node containing the system Processor. Its node contains the system Processor Unit, which in turn contains multiple functions corresponding to the drivers for the I/O hardware in the I/O node. The `channel_number` (or `transaction_ID`) field also multiplexes inbound messages from the Functions into the shared inbound queue. Function-independent interrupt dispatch and message delivery software in the Processor unit must use something like a `channel_number` to demultiplex these inbound messages and deliver them to the related Function driver or perhaps a `transaction_ID` to deliver the message to a specific instance of the function driver.

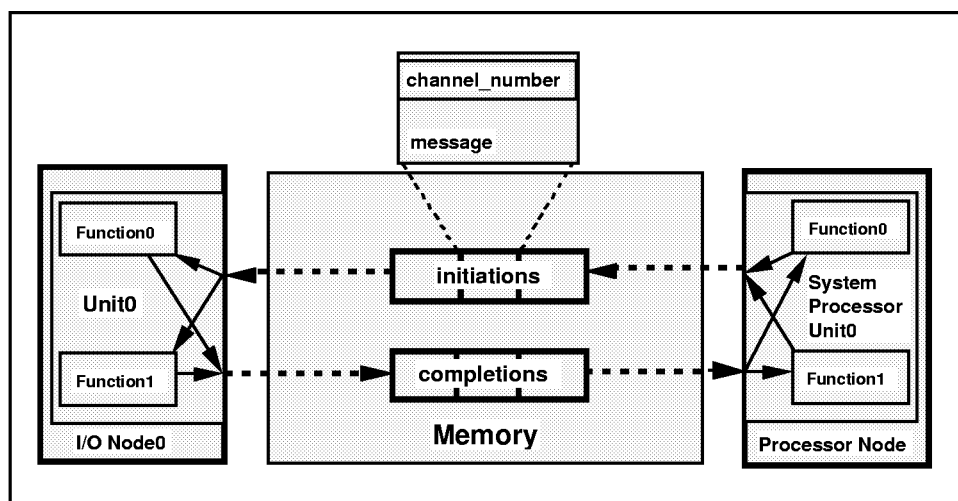


Figure 1.5—Multiple functions per unit

For example, I/O Node 0 could be a multi-function I/O card containing a disk function and a LAN function. The functions in the Processor node would be the corresponding drivers.

1.3.3.4 Shared inbound queues

The DMA models do not require a distinct inbound/outbound queue pair. There may be multiple queues in either direction (as seen in 1.3.3.5). But the Circular Queue and Dispatch List models do not support queues being shared by multiple I/O Units. For the Shareable List and Mailbox models that do support queuesharing, the inbound queues can be allocated based on the number of processors or the number of I/O priority levels, for example, rather than the number of Functions. If all inbound DMA messages were handled by one Processor, for example, one inbound queue could be shared by multiple I/O Units, as illustrated in figure 1.6. Again, the *channel_number* (or *transaction_ID*) fields in the inbound messages are used to route messages to the proper driver (and instance).

It is also not a rigid requirement that the inbound and outbound queues even use the same DMA model. However, it is a practical necessity to provide some convenient means for the closed-circuit flow of shared memory, including the memory used by the DMA message and data structures.

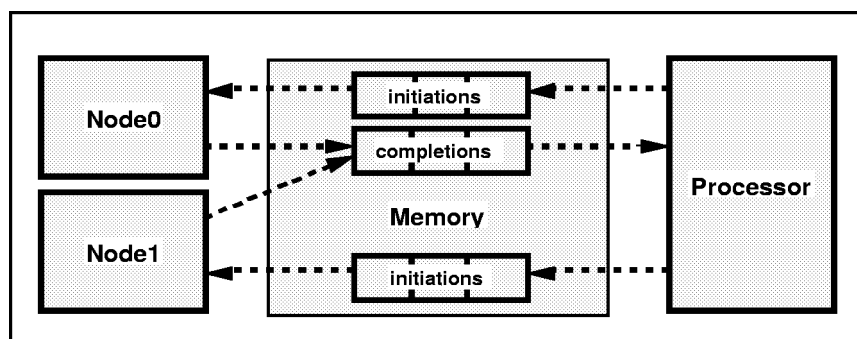


Figure 1.6—Shared inbound queue

1.3.3.5 Multiple channels per function

Subclause 1.3.3.3 illustrated the use of `channel_numbers` to multiplex messages from multiple Functions onto a single Unit queue, with one logical channel per Function. For all the function-to-node and function-to-queue cases described in 1.3.3.1 through 1.3.3.4, each Function will often require the backplane interface between a pair of *physical* entities to support a large number of *logically* separate channels or connections. Also, each logical channel will often carry event messages of varying priority and significance.

As noted above, the Function's application superimposes relationships on the message contents. These relationships may be very simple or enormously complex, varying greatly among applications. By studying the details of a realistic application, it will be more clear why a more primitive degree of DMA queue functionality simplifies the overall implementation.

Consider the example of an X.25 Function in figure 1.7. In this example, the user applications are aware of separate X.25 virtual circuits (connections), as is the X.25 Controller. The application entities in both the Processor and the I/O Function must have state records for each of a potentially large number of connections in order to operate.

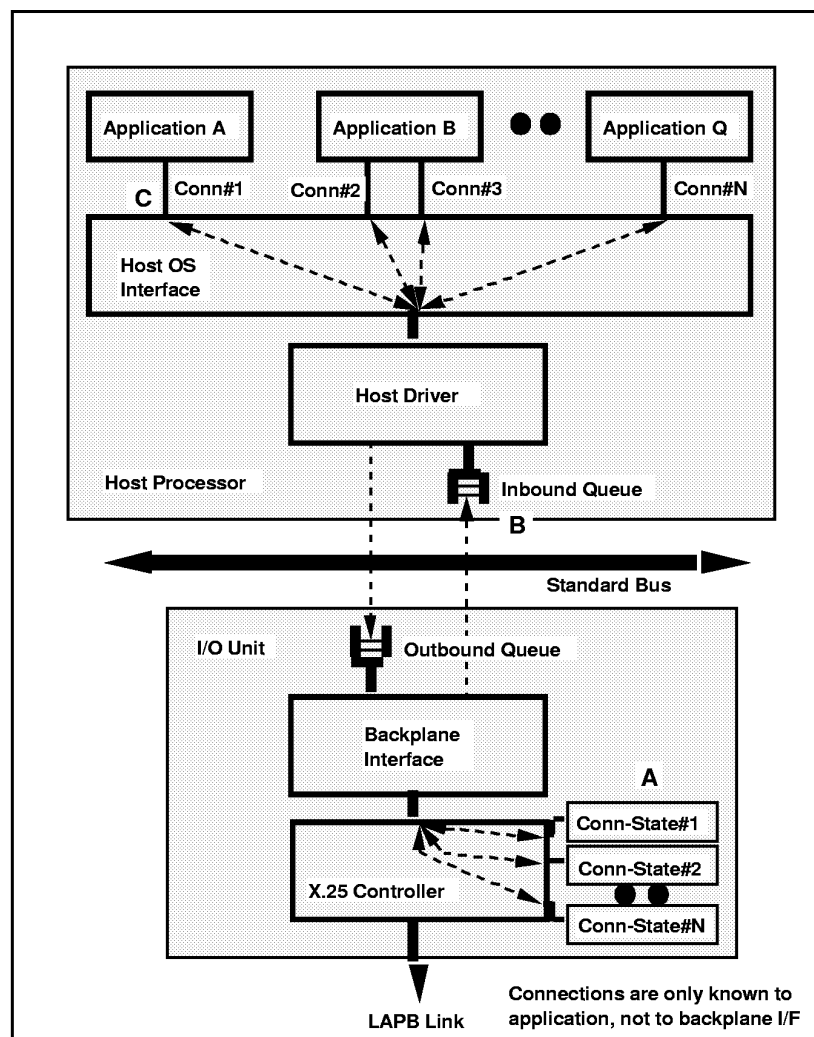


Figure 1.7—Logical connections: X.25 example

The relationships between the physical and logical components of a data channel can be illustrated as follows. When an inbound packet is received over the LAPB link, the X.25 Controller will retrieve a virtual-circuit number from the leading header bytes to determine which connection the packet is related to. This virtual-circuit number is mapped to a connection state record (“A” in figure 1.7). The connection state record contains error-control and network flow-control variables that are used to maintain the virtual circuit, as well as information that is used to link the virtual circuit to the Processor application. When the X.25 Controller has stripped the header bytes and finished processing the packet, it records this linking information (a `channel_number`), together with other packet-related parameters, in an inbound DMA message that is delivered by the backplane interface to the inbound queue (“B”). The Processor driver then reads the DMA message from the inbound queue and delivers it to the Host OS Interface. The Host OS Interface uses the linking information to determine which user interface data structure to queue the message on and which process to signal, indicating arrival of the message (“C”). In this manner, a single set of physical queues and related DMA hardware can be used to handle a very large number of connections. Notice that the DMA hardware itself was not required to know that multiple X.25 connections were being served.

Beyond the details of application-dependent channel multiplexing, there are even more compelling reasons for making the DMA hardware independent of long-term message queuing requirements. For each X.25 connection there can be events related to normal data delivery, expedited data, resets (which dump all data that is undelivered before the reset), a connection abort, intra-machine control and status, etc. Some of these events should be handled at varying degrees of priority, some are executed immediately, and some affect the contents of the queue itself. These operations vary greatly, depending on the application.

For example, an inbound X.25 Reset event requires all data sent in either direction before the event to be discarded without affecting data sent after the event (and without disrupting unrelated local command/status messages). This selective, application-dependent discarding would be very difficult to do if data were queued long-term in the DMA queues. Instead, these queues can be kept mostly empty and the reset event handled as follows. After receiving the Reset packet from the LAPB link, the X.25 Controller sends a Reset DMA message to the Host OS Interface. It then discards all internally queued data for either direction related to the state vector. When the Host OS Interface sees the Reset DMA message, it also discards all queued application data for this connection and sends a Reset Handshake message back to the X.25 Controller. The X.25 Controller watches the outbound DMA message stream for the Reset Handshake. It discards any outbound data messages it sees for this connection before receiving the Reset Handshake message, since these are known to have been sent before the Host OS Interface saw the Reset. Once the Reset Handshake message is received, the X.25 Controller knows that all data has been flushed through the queues in both directions and that his partner on the other side of the bus has completed handling the Reset event.

In summary, rather than complicate the inbound and outbound queues with these application-dependent requirements, these special queuing requirements will be provided by the applications themselves, tied to the channel state records. The DMA queue simply provides a high-performance means of delivering events, in their order of occurrence, across the bus between the Processor and the frontplane controller. Each interface event message would contain an *opcode* indicating the event type, a *priority* field, and a *channel_number* referencing the related connection. The significance of each event is sorted out on the other side of the interface after the message is delivered, affecting application-dependent scheduling and data structures. An event message only sits in the outbound queue long enough to get the frontplane controller's attention, and in the inbound queue long enough to get the Processor's attention.

Note that this DMA Framework does *not* restrict the number of queues per entity pair. Only one pair is normally needed. There may be additional queues, if required. For real-time applications where timing is extremely tight, for example, there could be a separate queue for high-priority events (see the discussion of priority handling in 1.3.4). Similarly, a separate inbound queue could be used for delivering error events to a different entity or handling them at a higher priority.

1.3.4 Message priority handling

Most applications can be satisfied by fair, non-prioritized message handling. When the consumer finishes the handling of each message, for example, it checks the queue of unserved messages and processes the oldest one.

A simple extension that supports most prioritized needs is to process the highest priority unserved message, rather than the oldest. This is analogous to sorting all newly delivered mail, then reading the oldest, most important letter. When the I/O Unit, for example, finishes the handling of each event, it can check for the arrival of new messages from the Processor, parse them, update a priority-ordered list of pending events (possibly deleting pending events that have been cancelled), and then continue with extended processing of the event at the head of the list. Priority is only distorted by the time needed to finish processing the current (possibly low-priority) event and sort the new messages.

Tighter priority handling requires preemptive scheduling. This is analogous to setting aside the letter you're reading whenever mail is delivered, sorting it, and reading a new one if it is more important than the one you were working on. For DMA, this would involve waking up a preemptive, high-priority process or interrupt service routine whenever an event message is delivered to the DMA queue. This process examines the event, updates application data structures (requeuing the event in an application-dependent manner), and possibly alters the scheduling sequence for application processing. When it is finished sorting new messages, it transfers control to the highest priority event-handling process now ready to run. Therefore, the event vectoring process can preempt the processing of the current event when necessary to handle an event at a higher priority. Priority is only distorted by the time needed to separate the low-priority message events from higher priority ones, which usually requires only a few instructions.

For extremely time-critical priority handling, separate message queues and interrupt priorities can be added. This is analogous to having a dedicated mailbox with a buzzer for urgent mail—when the buzzer rings, the new, urgent letter is read immediately. For DMA, each queue can then have its own preemptive event-vectoring and handling process. This would avoid the sorting time for lower priority messages. Sorting is done automatically when the sender selects the queue. But this is at the expense of providing, checking, and maintaining separate queues. Priority is only distorted by the time to switch process or interrupt contexts: the best that can be done without dedicating a Processor or I/O Unit to a given priority.

For all these levels of time-criticality, the Function must always ensure that event message space is available for higher priority messages.

2. References

This standard shall be used in conjunction with the following publications:

ANSI X3.159-1989, Programming Language—C.⁸

IEEE Std 1212-1991, IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses (ANSI).⁹

3. Document structure and notation

3.1 Document structure and conformance requirements

This document describes a number of criteria and problems to be dealt with in providing high-performance data transfer, together with techniques and overall frameworks for addressing them. Most of this information is necessary background for understanding the detailed DMA specifications and recommendations. Other clauses contain analysis, recommendations and examples of application.

⁸ANSI publications are available from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

⁹IEEE 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.

Four basic models of DMA are described: Circular Queue, Dispatch List, Shareable List, and Mailbox DMA. These are followed by optional enhancements, hybrids that combine the basic models, and a model for Device-to-Device DMA. Note that these DMA models do not interfere with one another. Different models may coexist in the same system.

Required specifications are located in clauses 5. through 9. for each individual DMA model, clause 11. for all DMA models in common, and 10.2 for the minimum format requirements for DMA messages. To avoid duplication and unintended divergence, specifications common to all models are collocated. So, for example, a Unit architecture based on the Shareable List model should conform with the specifications in clause 11. (which apply to all models), clause 7. (which describes the model itself), and 10.2. Model-specific requirements are given in table 3.1.

Table 3.1—Applicable specification clauses for each model

DMA model	Applicable specification
Circular Queue	Clauses 5., 10.2, 11
Dispatch List	Clauses 6., 10.2, 11
Shareable List	Clauses 7., 10.2, 11
Mailbox	Clauses 8., 10.2, 11

A given I/O Unit or Function architecture standard should specify the DMA model to be used, the structure and semantics of the DMA messages exchanged, the contents of the `unit_initialization_block`, methods of flow control to be used, how system memory is to flow in a closed-circuit fashion, how interrupts are to be used, etc., selecting from, and building on, the specifications and recommendations in this document. To the extent that messages from multiple standard Functions are multiplexed through shared queues for a given standard I/O Unit, or that messages from multiple I/O Units are multiplexed through shared queues (e.g., for an inbound Processor queue), common conventions for general message structure and flow control must also be shared among these Functions or Units. This document gives many recommendations in these areas, but specific decisions are left to follow-on activity to define standard Functions and Units.

3.2 Conformance levels

Several key words are used to differentiate between different levels of requirements and optionality, as follows:

expected. Describes the behavior of the hardware or software in the design models assumed by the DMA Framework. Other hardware and software design models can also be implemented.

may. Indicates flexibility of choice with no implied preference.

shall. Indicates a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other conformant products.

should. Indicates flexibility of choice but with a strong preference. “Should” is interchangeable with “is recommended.”

3.3 Technical glossary

The IEEE 1212 series uses many technical terms related to buses and interconnects. In addition to the terms used in the IEEE Std 1212-1991 CSR Architecture specification, the terms used in this document are described below:

3.3.1 32-bit supportive: Uses 32-bit addresses when accessing System Memory.

3.3.2 64-bit supportive: Uses 64-bit addresses when accessing System Memory.

3.3.3 address-invariant: A convention for defining the byte-ordering of DMA messages and data. The first byte in a contiguous set corresponds to the lowest order address, the second byte corresponds to the next address, etc. This is independent of how these addresses are sequenced onto a serial bus or assigned to physical positions and sequenced on a parallel bus. When different bus types are connected within a system, the bridge is expected to map the respective byte lanes and sequential positions to maintain address-invariance.

3.3.4 agent: An active component or bridge that acts on behalf of the real target for an action. For example, a DMA queue could be placed in a bus bridge in order to perform special operations when crossing address or protection domain boundaries.

3.3.5 always_swap: A bus transaction that atomically writes a new value to an address and returns the previous value (see table 7.2).

3.3.6 atomic transactions: Transactions that are indivisible with respect to other transactions. Other concurrent transactions will either see the state of memory as it was before the atomic transaction or after, but not a mixture. For example, a write4 transaction is atomic if no observer of the same address can see a mixture of old and new byte values, and two write4's for the same address are atomic with respect to each other if one overwrites the other without mixing bytes.

3.3.7 barrier transaction or write barrier: A transaction that ensures that all previously generated write transactions have the global appearance of having been written to memory. This is used before signaling another noncoherent unit, or one in a different coherence domain, that the data is available. In some systems, this is an explicit bus transaction that will be treated specially by the bus bridges (e.g., that may not return a response until all write buffers for the unit are flushed). For buses that delay the write-response until write bus transactions have been adequately completed, a separate barrier transaction is not needed since the effect of a barrier can be achieved by waiting for all outstanding write-responses.

3.3.8 big endian: A multibyte data value that is stored in memory with the most significant data byte through least significant data byte in the lowest through highest memory addresses, respectively.

3.3.9 big endian processor: A processor architecture that is optimized for the processing of big endian data values, as opposed to little endian data values.

3.3.10 block_descriptor: A cell in a block_vector containing a pointer to the first byte of a contiguous storage block, the size of the block, and a flag. The flag indicates whether the storage block contains data or an extension to the block_vector.

3.3.11 block_vector: An effective_length parameter and an array of block_descriptors referencing storage locations for application data or unit-dependent information. The referenced blocks are physically segmented but logically contiguous. The descriptor array may also be segmented and one flagged descriptor per array segment may be used to point to an extension of the vector. The effective_length parameter limits the transfer of data bytes. See 10.2.3.1.

3.3.12 channel: A logically independent data path between two Functions. Multiple channels can be used to reach different Functions or to represent independent instances of inter-unit communication (e.g., X.25 connections, I/O operations on different discs, datagrams to different network SAPs, etc.). Channels can either be provided by physically separate queues or by multiplexing a shared queue.

3.3.13 channel_number: A system-dependent, system-global value that is used by communicating Functions to designate a channel.

3.3.14 Circular Queue: A software array of message storage locations for which the last cell is logically adjacent to the first. It is a first-in-first-out (FIFO) queue where the producer leads the consumer through the array, wrapping back to the first cell when the end is reached.

3.3.15 compare_swap: A bus transaction that takes a test value and a new value as inputs, compares the test value to the current contents of an address and, if equal, atomically writes the new value to the address and returns the old (see table 7.2).

3.3.16 completion queue: : A DMA queue that is used primarily to pass I/O transaction-completion messages.

3.3.17 consumer: A unit that removes messages from a DMA queue.

3.3.18 CSR: A 4-byte control or status register in a unit's interface that is accessible via a system address.

3.3.19 CSR Architecture: Refers to IEEE Std 1212-1991, the parent document of this series. It specifies the overall architecture of a node on conformant standard buses and specifically the structure and use of its CSRs and standard ROM locations.

3.3.20 Dispatch List: A DMA model where the DMA queues are linked lists of dispatch_items. Each dispatch_item contains a pointer to the next dispatch_item and a message being passed to the consumer.

3.3.21 DMA: A direct memory access (or simply DMA) architecture is an optional capability of an I/O controller. After being started by the Processor, I/O controllers with DMA capabilities can access their commands, fetch data, and report status by accessing System Memory directly.

3.3.22 DMA Framework: Refers to this document: a description of the fundamental structures and protocols for the overall exchange of messages and data via shared memory. In this framework, a given DMA model is not a complete interface specification, but rather a skeleton. To it must be added conventions for message contents, semantics, and allowed sequences in order to define application-dependent I/O Unit and Function interface standards.

3.3.23 event: A semantic construct associated with a point in time that may result in an instance of processing or state transitions on the part of the receiver. Events are usually carried between entities by DMA messages. For example, an inbound DMA event message may indicate an asynchronous error requiring Processor attention.

3.3.24 externally visible: Actions or values that can be seen by other Units, as opposed to those that are unit-private. Externally visible conditions are specified by this document, unit-private conditions are not.

3.3.25 first-in-first-out (FIFO) queue: A data structure from which entries are removed in the same order that they were added. In this document, entries are added at the "tail" of the queue and removed from the "head."

3.3.26 flow control: A mechanism for signaling the producer when messages may or must not be sent. It is used to avoid overrunning the limits of the consumer, memory, queue-depth, or message-passing facilities.

3.3.27 Function or Unit Function: A logical component of a unit that operates mostly independently. A function is independent of bus interface and node topology: a unit has registers that are externally accessible in the bus address space, a function may or may not. For example, a multi-function node may contain SCSI controller, LAN interface, and terminal interface Functions that are accessed via a shared DMA multiplex Unit. A function may also be a software entity (e.g., a Function driver), especially when messages are used for Processor-to-Processor communication.

3.3.28 Function driver: A software function in the Processor that is a logical peer to a hardware or firmware function in an I/O Unit. For example, a SCSI Function driver is the target of an I/O completion message from a SCSI Controller Function on an I/O module.

3.3.29 Futurebus+: A name that refers to IEEE 896.1-1991 [B1] and companion 896 series standards. Futurebus+ defines a physically bussed backplane bus standards, which supports 32-bit and 64-bit physical addresses. IEEE Std 896.3-1993 [B2] discusses bridge and other architectural considerations.

3.3.30 inbound queue: A queue carrying messages from one or more I/O Unit Functions to a Processor.

3.3.31 initiator: The function that starts an I/O transaction-initiation/transaction-completion exchange by sending an initiation message to the responder.

3.3.32 initiation queue: A DMA queue that is used primarily to pass I/O transaction-initiation messages.

3.3.33 I/O Unit memory: Memory that is located in the I/O Unit. For most purposes in this document, the externally visible locations may interchangeably be in either the initial node space (4 kbytes) or unit-extension (or even the memory-extension) areas defined in IEEE Std 1212-1991.

3.3.34 Linked List: A software data structure composed of individually allocated memory objects, in which each item points to the next. It is used as a FIFO queue in this document.

3.3.35 little endian: A multibyte data value that is stored in memory with the most significant data byte through least significant data byte in the highest through lowest memory addresses, respectively.

3.3.36 little endian processor: A Processor architecture that is optimized for the processing of little endian data values, as opposed to big endian data values.

3.3.37 Mailbox: A register that maps write data to unit-internal message storage locations. The mapping between registers and message-storage locations is transparent to the one or more producers. A message is sent to a mailbox by performing an atomic block copy of the message bytes to the mailbox register. Following this message write, the mailbox hardware moves the mapping to the next message storage location to prepare for the next message write.

3.3.38 message: A value or set of values representing an interface event between functions. The term as used here is intended to be very primitive, not implying a particular structure or interface protocol unless modified by an appropriate adjective (like transaction-initiation message). A message can be arbitrarily simple (a signal) or complicated.

3.3.39 message_extension: An allocated buffer in System Memory containing items that either would not fit in the primary_message or that are only needed for unusually large messages.

3.3.40 message queue: A data structure and related procedures for passing a sequence of primary_messages from one or more producers to a consumer.

3.3.41 outbound queue: A queue carrying messages from a Processor to an I/O Unit, or from Processor to Processor.

3.3.42 primary_message: The part of the interface event message carded by the DMA mechanism itself. The primary_message may point to message_extensions.

3.3.43 Processor: A main system processor unit that executes operating system code and manages system resources. It is usually constrained on the number of CSRs it can devote to the functions of a given I/O Unit.

3.3.44 producer: A unit that adds a message to a DMA queue.

3.3.45 reply: See: **transaction completion.**

3.3.46 request: See: **transaction initiation.**

3.3.47 responder: The function that completes an I/O transaction-initiation/transaction-completion exchange by sending a completion message to the initiator.

3.3.48 sequence variable: A flow-control component involving a public location in System Memory or a CSR, holding the sequence number of the current message. The first message corresponds to sequence number one, etc. This sequence number is operated modulo the variable size (i.e., when the maximum value is reached, it rolls over to zero).

3.3.49 Serial Bus: A bit-serial interconnect defined by IEEE P1394 [B4] .

3.3.50 Shareable List: A DMA queue composed of a linked list of items. Each item contains a pointer to the next item and the message being passed to the consumer. Swap transactions are used to support shared access by multiple producers.

3.3.51 spin-wait: A condition during multi-unit synchronization where a processor remains in a tight loop, retesting a variable while waiting for the value to be changed by another unit. This condition may waste processor time and cost considerable bus bandwidth if the test variable must be read from system memory in another unit.

3.3.52 state space: Memory that is used to store the parameters, variables, work space, etc., related to an I/O transaction that is currently being processed.

3.3.53 System Memory: Read/write memory accessible by both the Processor and I/O Unit in the system address space. It is usually thought of as being in one or more third-party nodes, but it could alternatively be wholly or partially located in the communicating Processor or I/O nodes.

3.3.54 transaction, I/O: An instance of activity between Functions, usually composed of an Initiation and a Completion, although not necessarily bound one to one. A disk read and network data delivery are examples of I/O transactions.

3.3.55 transaction completion (reply): A reply generated by a function in response to one or more I/O transaction initiations. The completion returns status and sometimes data. In a disk read I/O transaction, for example, the completion returns the data and status from the disk and the function in the I/O Unit to the Processor.

3.3.56 transaction_ID: A value selected by an initiator to designate a given I/O transaction. It is included either explicitly or implicitly in a transaction-initiation message and returned in a transaction-completion message.

3.3.57 transaction initiation (request): A request generated by the initiator to start an action by the responder. An initiation message usually transfers a command and sometimes data. For a disk read I/O transaction, for example, the initiation transfers the address and command.

3.3.58 unit_initialization_block (UIB): : A contiguous buffer in System Memory used to pass Unit-dependent parameters to the I/O Unit during initialization or on command during normal DMA operation. The Processor writes the address of this block in a Unit-global CSR or passes it in a message, respectively.

3.3.59 user: The ultimate human interface or top-most application program. For example, text typed into a terminal interface can be referred to as “user data.” This document does not use the ISO Open Systems “layer-user” concept, in which each module in a vertical stack is the “user” of the adjacent, lower module.

3.4 Data structure notation

This specification shows all multibyte data structures in address-invariant format (monotonically increasing byte addresses). The DMA Framework supports unit architectures based on both little endian and big endian multibyte integer formats. Although this specification illustrates only the big endian data formats, the little endian equivalent representation can be generated, if necessary, by mapping bytes into the same address position in the little endian notation. For bit-level subfields, the equivalent little endian data structures can be derived by placing the subfields in the most-significant or least-significant bit positions as indicated by the text (see “N” in the lsb position in figure 3.2 and figure 3.3). Since there is widespread confusion related to how byte addresses overlay these two endian orderings, this subclause describes each.

3.4.1 Address-invariant (DMA) notation

The address-invariant notation used in this document shows structure fields in their locations in monotonically increasing address positions in a byte stream. In figures, this is folded, four bytes at a time, and scanned left-to-right, top-to-bottom. As an example, the format of a data block in memory could be specified for either big or little endian processors, as illustrated in table 3.2 and figure 3.1. The same data structure works for either processor, since the default ordering for multibyte integers is processor-architecture-dependent and compilers generally work with monotonically increasing addresses.

Table 3.2—Ambiguous C structure for big or little endian

```
typedef struct {
    octlet    add64;
    quadlet   count32;
    doublet   status;
    byte      info;
    byte      fill;
} example_struct;
```

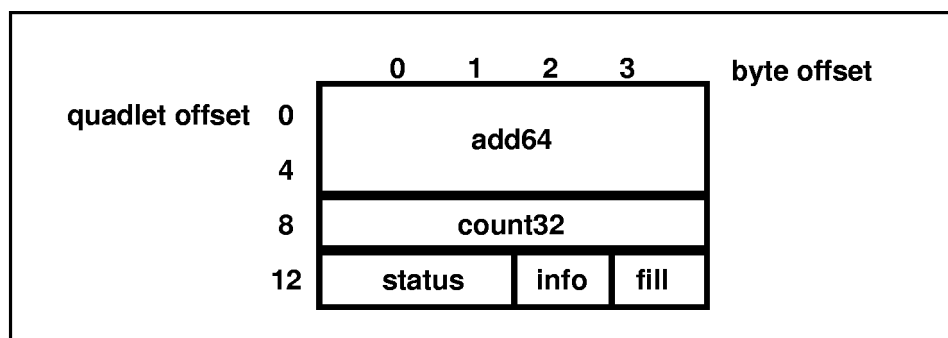


Figure 3.1—Address-invariant (DMA) representation

Notice that the position of the most and least significant bits in each subfield in figure 3.1, is indeterminate since it depends on the endian being used by the system. But the fields are defined to be at the addresses shown.

3.4.2 Big endian notation

For a big endian processor system, this ambiguous data structure is equivalent to the (processor-context-insensitive) data structure in table 3.3 and figure 3.2.

Table 3.3—C structure for big endian

```
/* big_xxx is an integer data type,
   0 is the most-significant byte */
typedef struct {
    big_octlet      add64;
    big_quadlet     count32;
    big_doublet     status;
    big_byte        info;
    big_byte        fill;
} example_struct;
```

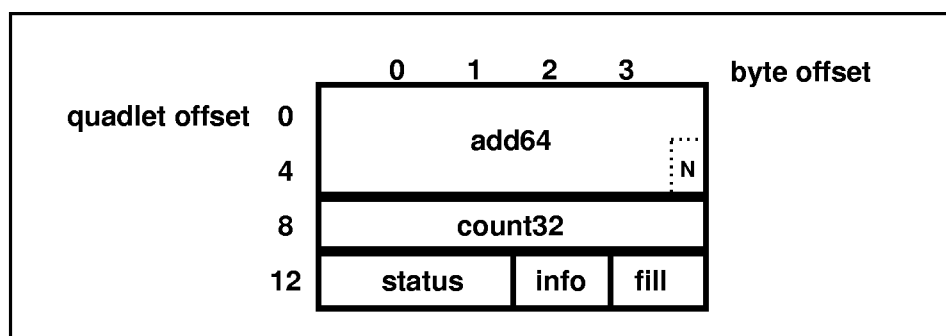


Figure 3.2—Big endian representation

Within the DMA Framework, the address-invariant notation happens to map onto the big endian notation byte-wise, except now the position of the most and least significant bits is known. This is shown by the “N” bit in the least significant bit of the add64 field in figure 3.2.

3.4.3 Little endian notation

For a little endian processor system alternative, the ambiguous data structure of table 3.2 is equivalent to the well-defined data structure in table 3.4 and figure 3.3.

Table 3.4—C structure for little endian

```
/* little_xxx is an integer data type,
   0 is the least-significant byte */
typedef struct {
    little_octlet    add64;
    little_quadlet   count32;
    little_doublet   status;
    little_byte      info;
    little_byte      fill;
} example_struct;
```

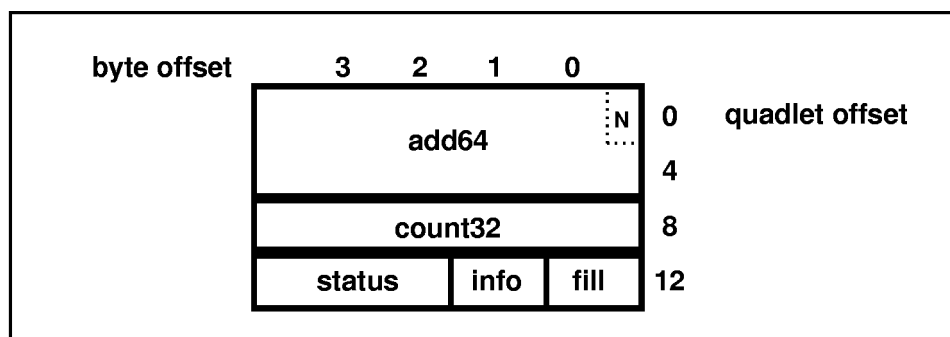


Figure 3.3—Little endian representation

A little endian data structure places the data at the same byte-address locations, but changes the significance of bytes within a multibyte object. This little endian alternative could be illustrated as shown in figure 3.3 (using a consistent, right-to-left-scanning, little endian notation).

Note that the address-invariance property can be used to determine the mapping of fields into the little endian representation. The data objects are always located at the same byte address, but that address sometimes appears at a different left/right location within a quadlet.

As mentioned above, for bit-level subfields, the illustrations in this document show the position for big endian systems, but the text takes precedence by indicating the true position. For example, for even-aligned objects, the least-significant bit of a pointer is used to indicate null when one. The position of this *null* flag indicated in this document is as shown by “N” in figure 3.2. For little endian systems, the actual position would be as shown in figure 3.3.

The little endian and big endian data structures are incompatible, in that a DMA Function designed to operate with integers of one endian order will not automatically work with data structures using the opposite endian order. However, DMA units may be designed to handle data structures of either endian order, using data-size-dependent byte-swap logic to convert multi-byte subfields. For example, a little endian I/O Unit in a big endian system can retrieve the proper value of count32 by extracting the bytes at offsets 8–11 and swapping byte 8 with byte 11 and byte 9 with byte 10. Note that no conversion is necessary when the I/O Unit and system share the same endian nor for fields that are uninterpreted by the I/O Unit (e.g., transaction_ID values).

Systems are expected to be configured with DMA Units that support the same endian order as the Processor(s).¹⁰ Some DMA Units may be designed to support either endian order (the order can be statically configured during system initialization). DMA Units may also be designed to support only one endian order, when use by Processors of the opposite endian order is not expected. The IEEE Std 1212-1991 CSR Architecture, which defines how nodes and units are identified and initialized, requires that the basic registers are always treated as big endian. This provides the necessary rendezvous mechanism for a configuring Processor to initially communicate with a configurable node to set the endian order to be used during on-line operation. Please see the related discussion in 11.3.3.

Similarly, some buses may be both 32-bit supportive and 64-bit supportive. For these buses, DMA units may be designed to support both address models (the address size can be statically configured during system initialization). DMA units may also be designed to support only one address model, when use with the other address model is not expected.

3.5 Node description (capitalization)

Description of the DMA models involves defining the actions of two or more entities that communicate via shared memory. From the viewpoint of the bus system and system performance, it is very significant that the DMA entities are often located in units on different nodes. A system Processor unit will commonly be located in one node, a System Memory unit in another, and the I/O Units and their Functions in additional nodes. To concisely refer to these entities and yet attempt to remind the reader of these relationships, capitalization will be used. Therefore, “Processor” will be synonymous with “the system processor unit in a separate processor node,” “System Memory” with “system memory unit in a separate memory node” (although shared memory could also be on either the producer's or consumer's nodes), “I/O Unit” with “an I/O Unit in a separate I/O node,” and “Functions” with an “I/O Function in a separate I/O node.”

3.6 Notation used for algorithms

The programming language C is used to help clarify the specification text for each DMA model. These code fragments are intended to be unit- and system-architecture-independent descriptions of functional behavior. They are unlikely to resemble an actual implementation and are not intended to imply one.

3.6.1 Notation for variables

All local variables in these algorithms are assumed to be unit-local (e.g., stored in I/O Unit RAM or efficient processor cache). For Processors, they are actually likely to be stored in System Memory, requiring bus activity to access.

Local variables use the same names defined for all DMA models in clause 11. or in the individual models. For example, interrupt_address and interrupt_value would be unit-local variables containing the items described in 11.5. The Unit's own CSRs are treated like local variables.

¹⁰This was also considered to be most reasonable since a microprocessor on a given I/O Unit may use an endian that differs from the system Processor as well as from other I/O Units. This avoids the need for both the Processor and I/O Unit to be prepared to convert endian orders and ensures that endian representations are consistent within a given system with no run-time switches. See 11.3.3 for a description of how the endian of a Unit is configured.

“A” is a predefined constant, giving the number of bytes in width of system addresses: 4 bytes for a 32-bit supportive system, 8 bytes for 64-bit supportive.¹¹

“PtrA” is a predefined typedef that contains a properly supportive system address value.

“Msg” is a predefined typedef that is an array of bytes containing the Unit-dependent fields of the primary_ message (see clause 10.) Other structure types will be defined by each DMA model description.

3.6.2 Notation for bus interface access

The prefix “B_” on a C function call will be used to indicate a bus access. The rest of the C function name will indicate the type of bus access or bus interface functionality that is being accessed. These names will be the same as the bus transactions listed in 11.1.1 (or, for special transactions, at the beginning of each DMA model description; e.g., table 7.1) or the DMA Controller functions described in 11.2. The body of the C function would involve whatever implementation-dependent actions are required to access this functionality. See the examples below.

The suffix on a bus transaction access name will indicate the data size in bytes. Usually this will be a number. The suffix “A” will signify an address-width transaction, either 4 bytes for a 32-bit supportive system or 8 bytes for a 64-bit supportive one. Nodes that can be configured to be supportive of either are expected to include parallel routines, one for each width. The suffix “N” will signify the largest configured allowable transfer size (often 64 bytes or 256 bytes).

Note that these suffixes refer to the *data* width, not the width of the address used to access the affected locations in System Memory. The address of the affected location will be the same for all transactions for a given system, 32 or 64 bits. The first parameter in these bus-access calls is the address. The second parameter is the unit-local data variable's address.

Examples of applying these prefix and suffix rules, assuming a 16-bit microprocessor, are as follows (referring to 11.1.1 and 11.2.1.2):

- “B_write16(sys_addr, loc_data)” would be a call to a bus interface chip function that might copy 16 bytes of unit-local data, starting at the node-internal address loc_data, into chip registers, load the affected system address value, sys_addr, into chip registers, and command the chip to do the writel6 bus transaction.
- “B_writeA(sys_addr, loc_ptr_value)” would be a similar call that would load 8 bytes for a 64-bit supportive system, or 4 bytes for a 32-bit supportive system, from the unit-local variable, loc_ptr_value, into chip registers and then command the chip to do a write8 or write4 bus transaction, respectively.
- “B_readN(sys_addr, loc_dest)” could be a function that copies the pointers into chip registers and commands the chip to do a read64 bus transaction.
- “B_get_block(sys_addr, loc_dest, num_bytes)” would be a similar access to the block-copy DMA Controller function to move a variable number of bytes into unit memory (referring to table 11.1).

This notation is used both for Processors and I/O Units in order to highlight bus activity. For Processors in cache-coherent systems, this may not accurately reflect the actual bus activity involved.¹²

3.6.3 Special macros and functions

Several special macros or functions will be used to simplify the notation and understanding:

¹¹Some bus standards attempt to accommodate 32-bit supportive nodes in a 64-bit supportive system by using the 32-bit wide format when possible for the lowest order portion of the address space and for addressing CSRs at the very upper end of the address space. In these systems, the 32-bit supportive nodes will continue to operate as they would in a 32-bit supportive system. The 64-bit supportive nodes are expected to always use 64-bit wide storage areas for general pointers, but will generate a 32-bit wide address in related bus transactions if the pointer value falls into one of the two appropriate ranges. The use of this “A” notation is not intended to imply that these nodes should always generate 64-bit addresses. The individual bus standards take precedence on the proper width to be generated for a given case.

¹²For example, the cache could snoop values when they're written to system memory. This would result in one trip of the data across the bus, when the notation used would imply two: one for the write and another for the read.

“ALLOC(item_address_ptr, item_size)” allocates a new chunk of system memory and returns a pointer to the new chunk. This chunk also may be allocated from a pool of resources that were passed to the unit by previous messages.

“ALLOCATE_MSG_RESOURCE S” represents unit-dependent activity to allocate message extension blocks, user data buffers, etc., related to a message. These resources may be allocated directly from system memory or from a pool of resources that were passed to the unit by previous messages.

“B_WRITE_BARRIER” returns when all previous writes have been completed. This is an abstract function that in actuality must encompass the various aspects of write-ordering discussed in 11.1. Since this is expected to be a relatively short delay, the algorithms are written such that execution resumes with the next line.

“DEALLOC(item_address)” returns a chunk of system memory to the system memory manager or to a unit-internal pool to be reused for the messages it sends. (A unit-internal pool is conceptually where memory from a transaction-initiation message may be held until a transaction-completion message is sent.)

“DEALLOCATE_MSG_RESOURCE S” represents unit-dependent activity to deallocate message extension blocks, user data buffers, etc., related to a message. These resources may be given directly to the system memory manager or deposited in a unit-internal pool of resources to be reused for the messages it sends.

“FILLBYTES(byte_ptr, fill_value, object_size)” fills any object with the given value. This is used mainly for initialization.

“IS_NULL_PTR(pointer_name)” is a macro that efficiently tests a pointer for null. Given the pointer conventions in this document, this involves a test for all-ones, or for fields where the *null* flag is used, simply a test for one in the least significant bit of the pointer value. The macro results in a value of TRUE if the pointer is null.

“EXIT_UNTIL_INTERRUPT” exits the routine and arranges for application-dependent rescheduling, such that the entire routine is repeated later. Since the wait for an interrupt is expected to be a relatively long delay, the algorithms are written to exit at this point, rather than loop.

“READ_UNIT_MSG(primary_msg)” represents unit-dependent activity to read and process the fields of the primary_message in unit-local memory, as well as any message-related memory blocks in system memory. These message-related memory reads involve bus activity via calls to B_get_block.

“WRITE_UNIT_MSG(primary_msg)” represents unit-dependent activity to produce and write the fields of the primary_message in unit-local memory, as well as any message-related memory blocks in system memory. These message-related memory writes involve bus activity via calls to B_put_block.

“<text>” indicates an implementation- or application-dependent action that cannot be captured with the other notation in this clause.

4. Background concepts

Following the discussion of background concepts common to all models, this clause describes four models for primitive DMA building blocks: Circular Queues, Dispatch Lists, Shareable Lists, and Mailboxes. The interface examples in annex A show that these primitive building blocks can be used in many ways to construct interfaces adapted for specific applications.

4.1 Miscellaneous components

The following concepts are applied in several places.

4.1.1 Expected interrupt mechanisms

The IEEE Std 1212-1991 CSR Architecture assumes that interrupt signals are implemented as write4 transactions. For each DMA model, the producer and consumer use interrupt signals to “wake” one another when they have taken some action with respect to the DMA queue, if this wake signal isn't already a byproduct of some other register write in the DMA algorithm. For each interrupt signal, the generator of the interrupt maintains an interrupt pair. These pairs may be statically configured or dynamically passed via DMA messages (e.g., in an I/O transaction-initiation).

An interrupt pair contains the address of an interrupt register and a value to be written to this CSR. The interrupt address (either 32- or 64-bit) determines the unit(s) that receive the interrupt (i.e., “wake-up” signal). The (32-bit) interrupt value is used by the unit owning the interrupt register (the receiver of the interrupt) to directly or indirectly dispatch the proper function. When the generator of an interrupt wants to get the other unit's attention, it writes the interrupt value to the interrupt location. What becomes of the interrupt value is dependent on the CSR's implementation, but two overall schemes will be described here: interrupt-bits and interrupt-queue.

The selection of the optimal interrupt scheme is related to the selection of a DMA model.

4.1.1.1 Interrupt-bits

For the interrupt-bits mechanism, like the broadcast interrupts defined by the IEEE Std 1212-1991 CSR Architecture, the interrupt value results in one or several of the 32 bits being set (see figure 4.1). At any point in time, the value stored in the CSR is the logical inclusive-OR of all the bits that have been set since the last time the owning unit serviced the CSR. Each bit is expected to correspond to a list of one or more DMA queue control structures that is set up at initialization time by the unit owning the interrupt register. To find the queues needing service, the interrupt service routine is expected to atomically read and clear the value in the CSR, then examine all of the queue control structures that correspond to each interrupt bit that was set in the register's value.

Once the queues needing service have been located, the interrupt service routine schedules the appropriate queue service routines. If the number of queues times their number of priorities is less than 32, then each bit may correspond to one DMA queue that can quickly be located. However, checking of queue-control structures is less efficient if more than one DMA queue corresponds to each interrupt bit.

The advantages of the interrupt-bits mechanism are that it is easy to implement, an indefinite number of interrupts can be queued, and the architectures for directed and broadcast interrupts are the same. The disadvantage is that the information content of the interrupt register is insufficient to efficiently identify large numbers of interrupt sources. Therefore, this model is most useful when the number of DMA communicants is small, or when the DMA models used support the sharing of queues among multiple producers (e.g., the Shareable List and Mailbox models).

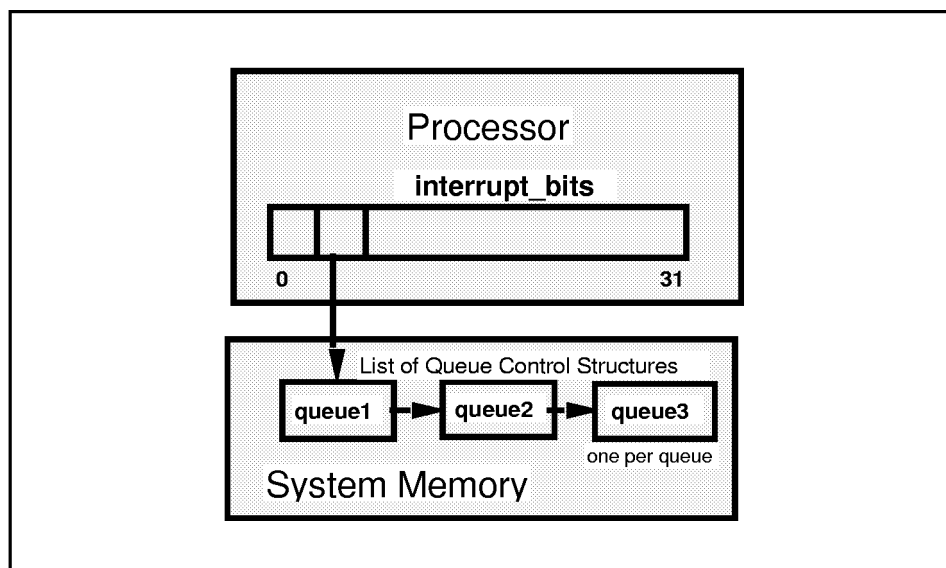


Figure 4.1—An interrupt-bits method of interrupt dispatch

4.1.1.2 Interrupt-queues

The interrupt-queues scheme functions like the Mailbox scheme described in 4.5.¹³ The write to the interrupt register maps onto a FIFO queue where each four-byte interrupt value is stored. This scheme allows the unit to vector directly to structures related to the interrupt's source.

The advantages of the interrupt-queues scheme are that a large number of interrupt sources can be uniquely identified with very low software overhead. Its disadvantages are the cost of the special interrupt queue hardware, and that such queuing schemes could theoretically overflow unit-local memory (without management), resulting in lost interrupts. This interrupt scheme is particularly useful when the number of DMA communicants is large and the DMA models used do not support the sharing of queues (e.g., Circular Queues and Dispatch Lists).

4.1.2 Methods of flow control

The need for flow control is intrinsic to the producer-consumer paradigm. If the producer sends messages faster than the consumer can accept them, eventually all the resources available to the message queue and/or consumer holding areas will be exhausted (e.g., all of System Memory for the Shareable List model). Therefore, every DMA application must provide some form of flow control, regardless of the model used. The description of the flow-control mechanisms used and the closed-circuit flow of memory is an important part of every standard unit specification that builds on this DMA Framework.

However, there is no need for redundant levels of flow control. If flow-control mechanisms are established to manage the utilization of the most scarce resource, unit operations can often be arranged such that the producer is stopped before there is any chance of other, more plentiful resources being overrun. For example, if application code in the initiator keeps a count of how many I/O transactions are outstanding with the responder, and if there is message storage space available in the initiation queue for more messages than the maximum number allowed to be outstanding, then the application-level flow control will avoid the need for flow control at the DMA message-queue level. This can be used to simplify the DMA algorithms.

¹³This type of interrupt is very close to Mailbox DMA. The difference is that in Mailbox DMA the value written to the register points to the message rather than a queue.

This clause briefly describes several flow-control techniques that can either be built into the various DMA schemes, or superimposed on them at the application level.

4.1.2.1 Flow-control handshakes

The simplest form of flow control is a direct handshake, or request-reply pairing of messages. For example, the Processor can monitor both the message-handling capacity of the I/O Unit and its own message queues. It can then delay sending a transaction-initiation message to the I/O Unit until all the resources needed are available in both directions. When the flow of initiations is stopped, an inbound transaction-completion message from the I/O Unit can then be used as a signal to initiate the next delayed I/O transaction.

A variation of this scheme is to dynamically allocate memory objects representing all the resources required for the initiation and completion, and then attach them to the transaction-initiation message. When the transaction-completion message returns, all resources can be reallocated to other uses.

Counts of usage can be kept per-channel, unit-global, or a combination of the two. Various levels of sophistication are possible to fairly distribute system resources.

A disadvantage of this scheme is that it is somewhat inflexible. One-way events are not handled well. Often extra messages must be passed, or “dangling reads” left posted, just to satisfy the flow-control scheme. This adds otherwise avoidable message-handling and interrupt overhead.

4.1.2.2 Flow control using sequence variables

A flow-control scheme can be built using two sequence variables and knowledge of the maximum consumer or message queuing capacity. One variable corresponds to the sequence number of the last message produced; the other to the number of the last message consumed. The number of messages unconsumed then corresponds to the difference (modulo the queue size) between the two values. When the difference is zero, the last message has been consumed. This operates much like the “take a number” scheme, where the two variables would be the next available ticket, and the “now serving number X” display. The number of people in line is the modulo difference between these values. Knowing the length of the line discourages new entries.

A change in the value of a sequence variable also can be used as a simple signal. By seeing the value change, the other party can tell that a message has been produced or consumed, and that the message-passing mechanism may now be serviced or resources reclaimed. In the analogy above, the server can infer that a customer has arrived by observing a change in the number of the next-available ticket.

4.1.2.3 Flow-control credits

This technique is similar to the sequence-variable method, except that the storage capacity for a given channel can be dynamically controlled by the consumer. This scheme is commonly used by network protocols. Each message carries a *credit* field to tell the other party how many messages can be sent before stopping. If there are no normal messages going to the other party on which the credit field can be piggy-backed, then a dedicated credit message can be sent.

This mechanism is very flexible. It does not tie inbound and outbound messages into request-reply pairs. But it is more complicated than the other schemes.

4.1.3 Software access to bus transactions

Buses conformant with the IEEE Std 1212-1991 CSR Architecture are expected to provide a number of powerful transactions (see 11.1 and 7.3). Some of these efficiently transfer a large number of bytes, while others perform special operations. Some of the DMA models require software access to these large or atomic bus transactions; all models are significantly more efficient when the optimum transactions are used. However, many of these bus transactions are not accessible to software since most existing processors have no equivalent instructions.

Most processors provide atomic access to word-sized memory locations (4 or 8 bytes), and many provide instructions to atomically handle double-word values (for floating point operations). But few provide instructions to directly access the larger Reads and Writes (16, 64, 256, N). These larger bus transactions are commonly used by the cache system, mostly transparently to software, and by I/O Unit block-copy engines. Instructions may be provided to flush a line of cache to its related address, but rarely to another target memory or CSR address.

Until direct software access to these bus transactions is provided in future processors, one potential approach is to use a hardware “staging area,” preferably on the same node as the processor. (In some cases, the bridge to an I/O bus may be a more appropriate location.) This staging area could appear to the processor to be a special range of node-internal memory or registers, as shown in figure 4.2. Most of the locations would be used to hold the operands for a bus transaction while others would be used for control. The processor would write to the operand locations to set up a bus transaction, write to the control register to cause it to be generated, then read the results from the operand locations.

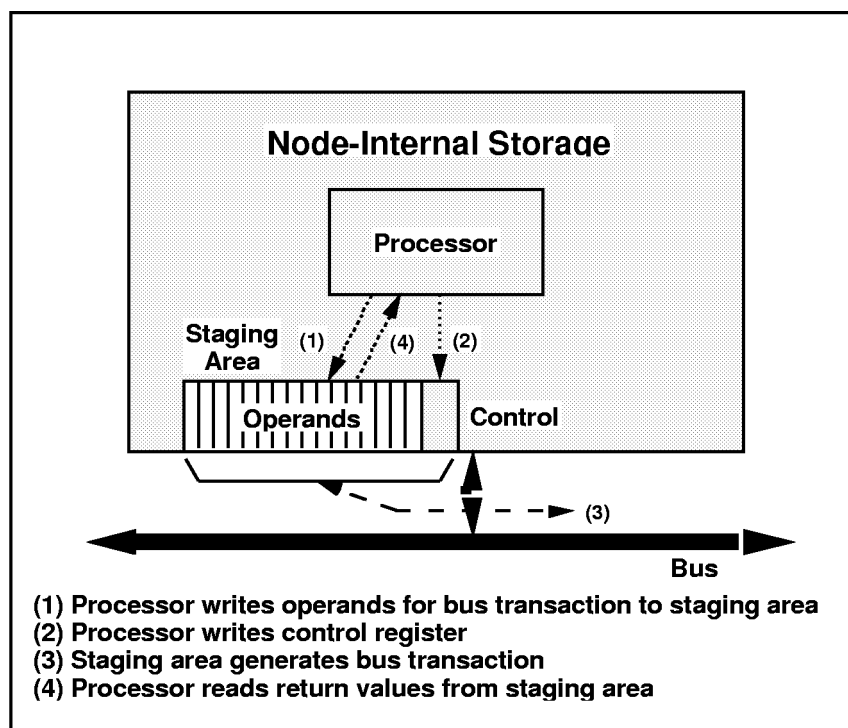


Figure 4.2—Staging area for supporting bus transactions

For example, a write64 could be generated by using write4 or write8 instructions to write the data values and target address to the operand locations, then a “write64” opcode to the control register. A compare_swap8 could be generated by using write4 or write8 instructions to write the target address, new value, and test value to the operand locations, then write a “compare_swap8” opcode to the control register, followed by read4's or a read8 to retrieve the return value.

The disadvantages of this concept are the hardware cost and backward-compatibility problems for existing processor boards. Another problem is synchronizing access to the staging area by multiple processes. An interrupt service routine, for example, may need to save and restore the operand registers to avoid corrupting their use by a background process. Still, these are node-internal design issues that may well pay their way by improving overall system performance.

4.2 Circular Queue

The Circular Queue model is based on a simple, widely known data structure called a circular queue. Circular queues are typically implemented as an array and have two indexes associated with them, a `producer_index` and a `consumer_index` (often called the head index and tail index). They are operated in FIFO fashion. A given queue is only used to pass messages in one direction, from a single producer to a specific consumer.

A circular queue can be thought of as an array in a circle, rather than a straight line. In this way, as items are added and removed from the queue, the head will continually chase the tail around the array. At different times the queue will occupy different parts of the array, but will never run out of space unless the array is fully occupied. The queue item storage cells around the circle are numbered from 0 to $n-1$. When the index is moved past $n-1$ it starts over again at 0.

There are several ways to implement circular queues. The methods differ in the way that the empty and full queue cases are detected. The method described here always leaves one empty position in the queue. Leaving one empty position, removes the ambiguity as to whether the queue is empty or full. The queue is empty when the producer and consumer indexes are equal. The queue is full when the producer_index is exactly one position behind the consumer_index. See figure 4.3.

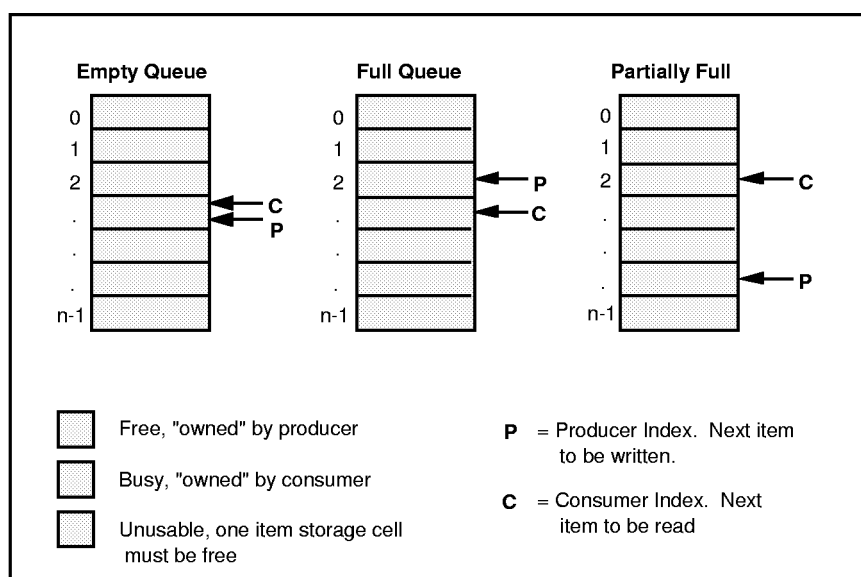


Figure 4.3—Circular queue computer science concept

If the queue is not empty, then all queue items from the one indicated by the `consumer_index` to the item one behind the `producer_index` are "owned" (can be accessed) by the consumer. All queue items from the one indicated by the `producer_index` to the item two behind the `consumer_index` are "owned" by the producer ("two" because of the reserved empty item). Based on these assumptions, here is a simple algorithm for adding items to the queue and for removing them (see also table 4.1). When adding an item to the queue, the queue item indicated by the `producer_index` is filled and `producer_index` is incremented (modulo n). When removing an item from the queue, the queue item indicated by the `consumer_index` is emptied and the `consumer_index` is incremented.

The algorithm in table 4.1 becomes even simpler if higher layers provide flow control (e.g., if the size of the Circular Queue is larger than the number of messages potentially outstanding). Then, the `consumer_index` can be private to the consumer, and the producer can simply store each new message and update the `producer_index`. The lines marked "opt" in the C producer routine in table 4.1 can be removed.

Table 4.1—Circular queue algorithms

Producer()	Consumer()
<pre> { u_char next_p; next_p = (p + 1) % n; if (next_p == c) { /*opt*/ <queue is full>; /*opt*/ } /*opt*/ else { /*opt*/ <fill p>; p = next_p; } } </pre>	<pre> { if (p == c) { <queue is empty>; } else { <empty c>; c = (c + 1) % n; } } </pre>

When applied to DMA, the Processor writes to an I/O Unit CSR to send it a signal when necessary that it has processed one or more messages in the Circular Queue. Similarly, the I/O Unit sends an interrupt to the Processor to signal it when necessary that it has affected a queue.

An attractive feature of Circular Queues is their synchronization mechanisms for memory access. Only the producer is allowed to write the producer_index as it fills the queue. Only the consumer is allowed to write the consumer_index as it empties the queue. Therefore, there is no confusion about which queue items are in use (filled) and which items can be read or written by each party. No semaphores or special bus transactions are needed between the producer and consumer, since there is no danger of one party overwriting the other's values when the queues are used properly.¹⁴ This is fairly robust. Only the producer needs to write into queue item storage cells. For cached Processors with noncoherent I/O Units, this means only its own previously written values will be overwritten (by the same value) as a side-effect of the cache flush during the write of a new item. Queue items need not even be a multiple of the cache line size.

Another feature is block access to messages. If the consumer falls behind the producer, several messages may be retrieved in one block read (e.g., transfer to Unit memory, or a Processor forced cache miss). By reading one index, the consumer knows how many unread messages are pending, and the producer knows how much space is available in the queue.

However, these synchronization mechanisms do not extend easily for use by multiple producers. So Circular Queues are hard to share.

Other problems with the scheme include the need for a potentially large and fixed contiguous block of memory for item storage. It is difficult to expand this storage on-the-fly by stretching the block.¹⁵ When idle, item-storage memory cannot be used for other purposes. Queue items are fixed-size, which can be a problem if messages are variable-length. These problems can be overcome by using a queue item only to point to variable-length items. But this would cost additional memory manager overhead and additional block-copy operations. But since the queues only hold messages until the consumer can notice them (i.e., they contain no long-term storage), the size required may not be significant for many applications.

¹⁴Note that only synchronization between a single producer and consumer is being discussed here. Multiple producer processes, for example, may need to use semaphores among themselves in order to act like a single producer.

¹⁵See annex E.

The reasons for using indexes instead of pointers include: They are more compact than pointers—normally only the least significant byte will be used. There is only one size of concern (i.e., the width of the index field) regardless of the width of address values. Indexes are easy to update atomically. A shared index value can easily be translated by a Function driver into a cell's virtual address, and can also easily be translated by an I/O Function into a cell's physical address. They are convenient for keeping parallel copies of module-internal state information related to each queue item (as illustrated in annex A). Since using an index implies using fixed-size queue items, the consumer always knows how many bytes to block-copy to pick up one or more items.

4.3 Dispatch List

The second DMA model is based on the cornerstone of all CS concepts: the linked list. This subclause describes the particular use of these structures.

The data structures in this scheme are operated in FIFO fashion: items are only added to the back and removed from the front. Items are not added or removed at other positions in the list (e.g., in priority order). Also, in this case, FIFO only refers to the order of production and consumption of message-passing. It does *not* imply that the events represented by these message (e.g., disk reads) will necessarily be completed in this order.

Dispatch Lists are conceptually simple but present a number of opportunities for race conditions and hazards with respect to the manipulation of their pointers. These problems exist between the producer and consumer, even when the list is not shared by multiple producers or consumers. This subclause describes the best algorithm arrived at that does not require the use of special (swap) bus transactions, and the next describes an algorithm that should be used when these bus transactions are available.

This Dispatch List algorithm assumes that there is a single producer and a single consumer. However, higher level multiprocessor synchronization mechanisms can be used to synchronize queue accesses to maintain the appearance of a single producer or consumer.

A Dispatch List message queue is a chain of dispatch-items, each containing a pointer to the next dispatch_item, and a (small) primary_message that usually points to a message extension, as figure 4.4 shows.

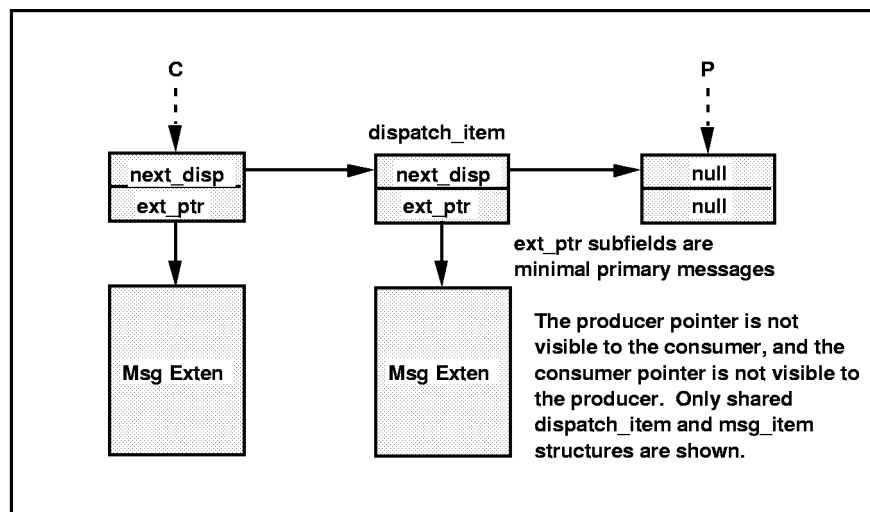


Figure 4.4—Dispatch list queue structure

Unlike the Circular Queue and Shareable List schemes, the consumer pointer (head of the list) and producer pointer (tail) are not visible to the other party. They may therefore be located in efficient unit-internal registers or cache. The producer and consumer rendezvous at the dispatch_item at the tail of the list. There is always at least one dispatch_item on the list. So a stable list contains n messages and $n+1$ dispatch_items.

Dispatch_items belong to the producer as long as the next_disp field is null. When the producer adds a new dispatch_item by making the next_disp field of the old tail point to it (therefore, making it non-null), the old tail becomes the property of the consumer.

Figure 4.5 illustrates the specific steps in adding a message to the queue. The producer allocates a new message extension and fills in the contents. It then makes the primary_message field of the old tail item point to the new message.¹⁶ Then a new tail dispatch_item must be allocated and initialized to null. Finally, the next_disp field of the old tail is set to point to the new tail. This final step transfers control of the old tail dispatch_item and the related new message extension, from the producer to the consumer. The new null tail item will remain on the list when the list is emptied by the consumer.

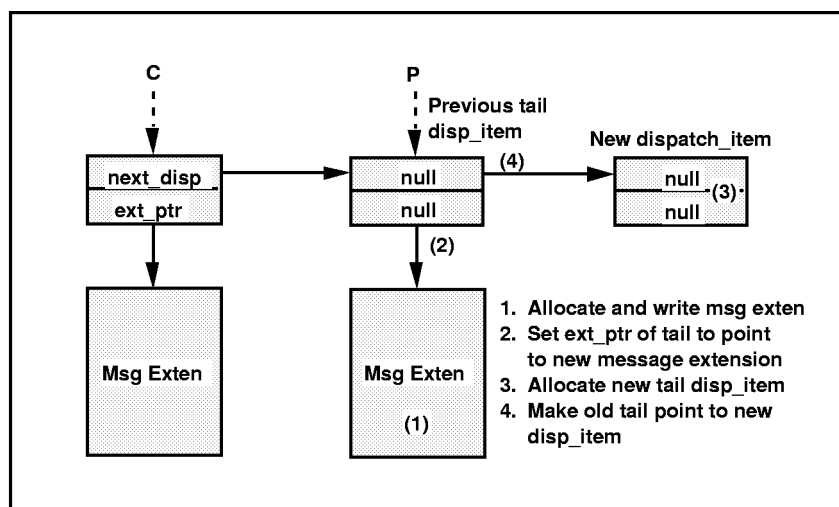


Figure 4.5—Adding a message to a linked-list queue

To remove an item from the queue, the consumer checks the next_disp field of the dispatch_item at the head of the queue. If it is non-null, it consumes the dispatch_item and the attached message extension, making the next dispatch_item be the head of the list. When the consumer finds a dispatch_item with a null next_disp field, the queue is empty. It must then wait for a signal from the producer to re-read the tail dispatch_item to retest it for ownership and availability for consumption.

The tail dispatch_item is used as the rendezvous site between the producer and consumer so that a given field has only one writer at a time, and so that only one field is involved in the transfer of ownership between producer and consumer. Other algorithms that use the head or tail pointer as a rendezvous site would require some means of synchronization between the producer and consumer to protect write-access to the pointers and dispatch_items. For example, there is a race between the consumer taking the last message in the queue and the producer adding another. If the head pointer were a rendezvous site, the consumer would be trying to set the head pointer to null when the producer is trying to link a new message to the one consumed. The producer could be writing to the link field of a “dead” dispatch_item or could

¹⁶It would be very helpful to the consumer if additional event-dispatching information were included in the primary_message as well. In this way, the consumer doesn't need to read the msg_item until the consumer is ready to process the related event. See 10.3.4 for a prioritized list of information to include as space permits.

fail to update the head pointer to reference the new `dispatch_item` (thinking that the head still pointed to the recently consumed message).

These race conditions are resolved by the careful use of swap transactions in the Shareable List scheme in 4.4.

When applied to DMA, the Processor writes to an I/O Unit CSR to send it a signal when necessary that it has processed one or more messages in the outbound queue. Similarly, the I/O Unit sends an interrupt to the Processor to signal it when necessary that it has affected the inbound queue.

Note that for this scheme the allocation of the message extension and `dispatch_item` are one step out of phase. Newly allocated `dispatch_items` are not linked to the message that caused their allocation. Also, the overall message-passing paradigm must make some arrangement for `dispatch_items` to be recirculated. Somehow there must be a closed-circuit allowing `dispatch_items` to flow back from consumers to producers. Since the `dispatch_items` and message extensions are not related, `dispatch_items` may need to be globally interchangeable. In other words, the producer will need to get `dispatch_items` back, but it *cannot* care about which ones it gets in exchange for the ones it gives away. For example, a `dispatch_item` cannot repeatedly be suballocated from memory that is contiguous with a message extension. The time when the message extension becomes available for reuse is unrelated to the time when the `dispatch_item` becomes available.

One way to return `dispatch_items` would be to provide one transaction-completion message for each transaction-initiation message (paired request-reply). This would allow the responder to replace one `dispatch_item` when it normally signals that the initiation message memory is available for reuse. But this may result in interrupt and message-handling overhead that is otherwise avoidable when multiple completions can be indicated in a single completion message. Another way would be to include pointers in the transaction-completion message for passing extra `dispatch_items` back to the producer. A third way would be to provide a separate reverse-direction queue that is only used for the return of excess `dispatch_items`. The only time a wake-up would need to be arranged for this queue would be when the producer is waiting for a `dispatch_item` to become available.

As with the other DMA schemes, a worst-case number of `dispatch_items` can be preallocated and stored in a free list that is protected by higher-level flow control. Then, when everything else is ready for an event message to be sent, a `dispatch_item` would be guaranteed to be available. Alternatively, a Processor's I/O transaction initiation could be delayed until a `dispatch_item` is available for dynamic allocation.

4.4 Shareable List

This subclause contains a linked list algorithm that should be used instead of the scheme in 4.3 when swap transactions are available, both for multiple-producer and single-producer applications.

It would often be attractive if queues could be shared by multiple, independent producers.¹⁷ For DMA applications, it may help the system design if multiple I/O Units could share an inbound queue (especially when there are many Units). This would allow the Processor to avoid the need to poll or queue interrupt vectors in order to know which inbound queue has an item ready for service. Similarly, in a multiprocessor system, it may be somewhat desirable to allow Processors to pass outbound messages independently to an I/O Unit.¹⁸ If external means of synchronization are provided, then the Dispatch algorithms from the last subclause could be used to share a queue.

The swap or locked transactions are defined on all buses conforming to IEEE Std 1212-1991, although they are only optionally available for some of these buses. These transactions, which apply to uncached memory, provide a basis for sharing queues among independent entities.¹⁹ Figure 4.6 shows how the `always_swap8transaction` can be used to provide synchronization among multiple producers. Note that the times in figure 4.6 correspond with those in the

¹⁷The use of multiple consumers is discouraged because of the unavoidable spin-waits involved in accessing the head pointer.

¹⁸This is not needed where Processors must synchronize among themselves anyway with respect to I/O Unit state (e.g., transaction-handling capacity).

¹⁹For a general discussion of software algorithms for queue-sharing using these primitives, see [B5].

scenario in table 4.2. Always_swap8 is effectively an atomic write8 that also atomically returns the 8-byte value that was overwritten. Locked transactions can be used to synthesize an always_swap8 by providing an atomic read8-write8 pair. In this example, two producers are trying to add an item to the queue simultaneously. All producers share the producer_ptr. The consumer_ptr is treated by the producers as though it were the next_item field of a list item. It is only written by the producers when the producer_ptr points to it. At all other times, it is only accessed by the consumer.

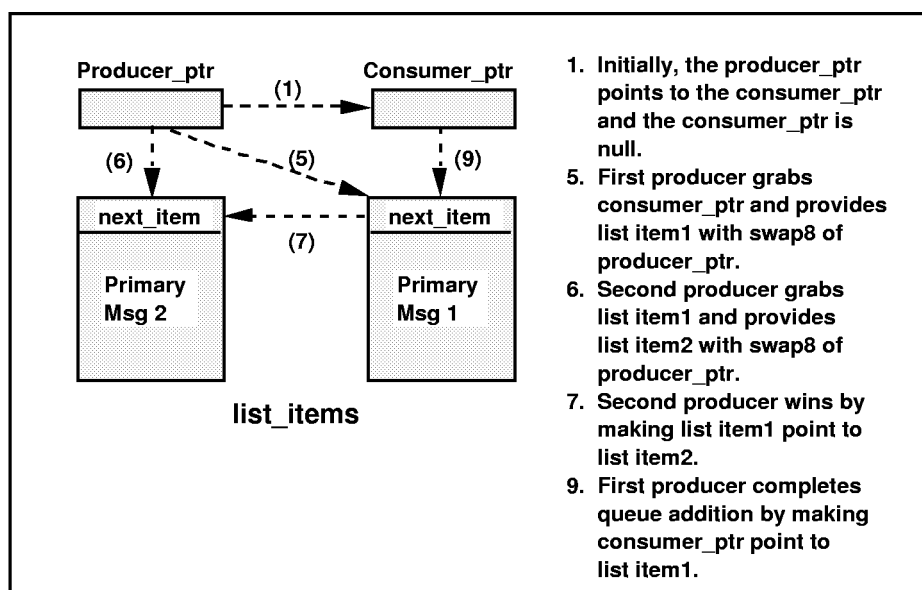


Figure 4.6—Always_swap8 use by multiple producers

Note that the entire messages are usually included directly in each list item in this model instead of needing an indirect message_extension as in the Dispatch List model: Shareable List items are not as limited for space as are dispatch_items.

Both producers allocate and initialize their list_Items at Times 1 and 2. Around Time 3, both producers try to add their messages to the list. The first producer (Producer1) accesses the producer_ptr first, takes control of the consumer_ptr and simultaneously provides a pointer to list_item1 by use of the always_swap8 bus transaction. When the swap value returns to Producer1, it effectively gives him permission to write to the consumer_ptr. Notice that the always_swap8 transaction always atomically leaves the producer_ptr pointing to the correct item for the next producer, so subsequent producers are never delayed. At Time 6, the second producer (Producer2) takes control of list_item1. The figure and scenario illustrate what happens when Producer2 finishes first. Each producer attempts to pass control to the consumer by setting the next_item field of the previous list item (the one returned by the swap) equal to their contributed tail item. If later-arriving producers finish first, then the queue data structure will momentarily be in a piecewise-continuous state. This will be corrected when the earlier producers (e.g., Producer1) finish by writing their next_item values, completing the missing links. The consumer will not be able to reach the items on the other side of a missing link (e.g., the link marked “9” in the figure). The worst that could happen is that Producer2 wakes up the consumer at Time 8, allowing it to see a null in consumer_ptr or a next_item field. The consumer will conclude that the list is empty, ignore both list_item1 and list_item2, and go back to sleep. But then at Time 10, Producer1 would make both new messages available and reawaken the consumer. (This implies that the wake-up signals for both producers must be given the same priority.)

Normally, the consumer only needs to examine the consumer pointer. If it is not null, it follows the pointer to the first list_item on the list. If the next_item field is not null, then no race condition is involved and it can simply process the command and set the consumer_ptr equal to the next_item field of the message consumed, removing the first message from the list. If the next_item field is null, then a race condition may exist.

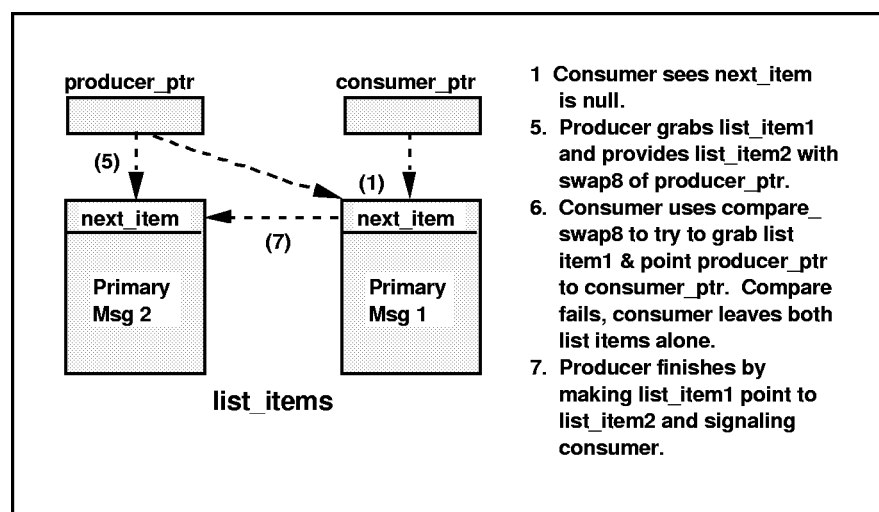
Table 4.2—Simultaneous access by two linked-list producers

Time	Producer1	Producer2	producer_ptr	consumer_ptr	list_item1. next_item	list_item2. next_item
1	Allocates list_item1	Allocates list_item2	&consumer_ptr	NULL	X	X
2	Fills in list_item1	Fills in list_item2	&consumer_ptr	NULL	NULL	NULL
3	Swap8(&prod_ptr, &list_item1)		&consumer_ptr	NULL	NULL	NULL
4		Swap8(&prod_ptr, &list_item2)	&consumer_ptr	NULL	NULL	NULL
5	Wins: returns &cons_ptr		&list_item1	NULL	NULL	NULL
6		Loses: returns &list_item1	&list_item2	NULL	NULL	NULL
7	{Takes a little more time ...}	Sets list_item1.next_ item = &list_item2	&list_item2	NULL	&list_item2	NULL
8		Signals consumer	&list_item2	NULL	&list_item2	NULL
9	Sets consumer_ptr = &list_item1		&list_item2	&list_item1	&list_item2	NULL
10	Signals consumer		&list_item2	&list_item1	&list_item2	NULL

When the consumer sees a null consumer_ptr or a null next_item field in a command, it may be racing a producer who is trying to add another item to the tail of the list. To check for this, the consumer does a compare_swap8 on the producer_ptr. The input parameters of the compare_swap8 transaction include the pointer to the current tail list item as the test value and the consumer_ptr as the conditional new value. The transaction atomically reads the current contents of producer_ptr and if the test value matches, sets it to the new value. It then returns the value read. So if the producer_ptr were pointing to the tail list item being processed by the consumer at the time of the compare_swap8, it will now be pointing to the consumer_ptr. If the value returned by compare_swap8 is also a pointer to the current tail_item, then the consumer will know that the swap succeeded, and that the list is now empty. The consumer will do a second compare_swap8 to see if the consumer_ptr still points to the old tail item and, if so, set it to null. This sets the list back to its initial condition. If either compare_swap8 fails (which the consumer detects by getting an unexpected return value), then a producer is in the process of adding a new item to the tail of the list. If the compare_swap8 on the producer_ptr fails, then the consumer leaves the existing tail alone and waits for the wake-up from the producer before processing both the existing tail and newly added tail items. This case is illustrated in table 4.3 and figure 4.7. (Note that the times in figure 4.7 correspond with those in table 4.3.) Since the compare_swap8 on the consumer_ptr is only an attempt to set it back to its initial condition, if it fails no further action is needed by the consumer since the producer has removed the need for doing this by already setting it to its next value.

Table 4.3—Simultaneous access by producer and consumer

Time	Producer	Consumer	Producer_ptr	Consumer_ptr	list_tem1. next_item	list_tem2. next_item
1	Allocates list_item2.	Sees list_item1.next_item==NULL	&list_item1	&list_item1	NULL	X
2	Fills in list_item2		&list_item1	&list_item1	NULL	NULL
3	Swap8(&prod_ptr, &list_item2)		&list_item1	&list_item1	NULL	NULL
4		Compare_swap8(&prod_ptr, &list_item1, &cons_ptr)	&list_item1	&list_item1	NULL	NULL
5	Wins: returns &list_item1		&list_item2	&list_item1	NULL	NULL
6		Loses: returns &list_item2	&list_item2	&list_item1	NULL	NULL
7	Sets list_item1. next_item = &list_item2	Consumer thinks list empty; waits for signal	&list_item2	&list_item1	&list_item2	NULL
8	Signals consumer		&list_item2	&list_item1	&list_item2	NULL

**Figure 4.7—Swap use for producer/consumer synchronization**

This description also illustrates the negatives of Shareable Lists:

- 1) All sharing entities must be of trusted reliability or else a failure could corrupt the shared data structure. For example, if a producer or bus bridge experienced a power failure before writing the next_item field, the list would be segmented.
- 2) Fast producers and consumers may be delayed by a slow producer sharing the queue.

These problems would exist for any scheme that manages shared data structures via semaphores or locks.

Figure 4.8 shows a useful producer optimization for cases where there are multiple items to enqueue at once. The producer can first link multiple items together off to the side. Since they are at this point completely under its control, atomic transactions or even reads of next_item pointers from system memory are not necessary. When the mini-list is complete, the producer adds the mini-list to the queue as though it were one virtual item. Specifically, the producer uses always_swap8 to make the producer_ptr reference the last item in the mini-list and retrieve a pointer to the old tail. The old tail is then made to point to the first item in the mini-list.

Since the list is shared, care must be taken not to abuse the mini-list optimization. Adding too long a mini-list to the queue will keep an idle consumer waiting longer, rather than operating in parallel with the producer (especially if the producer waits to see if more items are coming). It can also distort fairness among producers while the mini-list is being consumed.

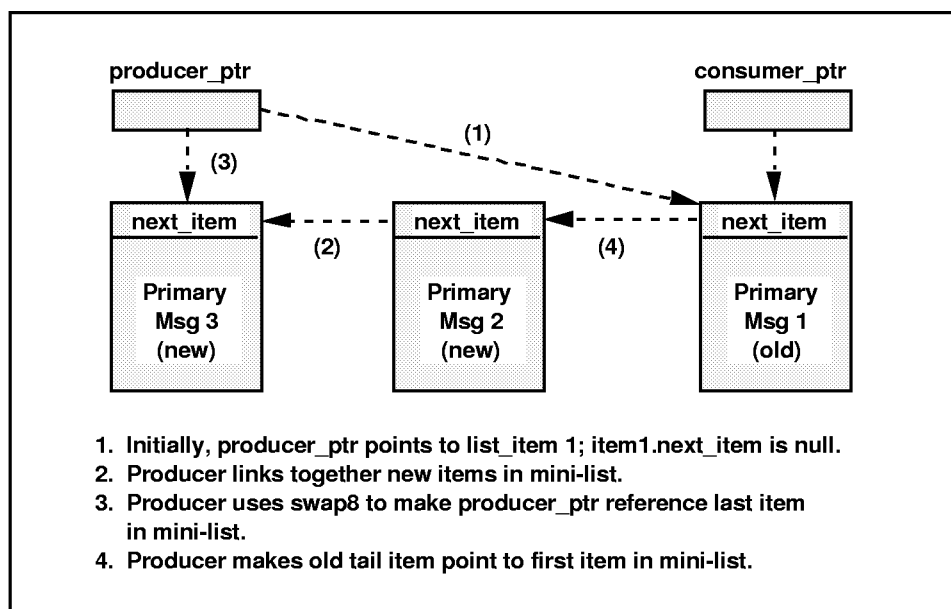


Figure 4.8—Mini-list producer optimization

4.5 Mailbox

Mailboxes are primarily designed to handle a large number of producers gracefully. In the previous schemes, the producer knew where each message was stored in System Memory, as either an element in a linked list or a cell in a Circular Queue. To solve the problems of multiple producers contending over message space, either a separate inbound queue was used for each producer or else special algorithms and compound bus transactions were used. A mailbox hides the message storage area from the producers. As defined by IEEE Std 1212-1991 and this document,²⁰ to send a message, the producer simply uses an atomic write to a mailbox register. This register maps the write data

²⁰There are other mailbox schemes where “sending to a mailbox” does not involve a register or an address in the system address space. The mailbox message storage area is also often hidden from both the sender and receiver.

through to message storage locations inside the unit, as figure 4.9 shows. Normal bus arbitration provides synchronization of mailbox access among multiple producers. The unit owning the mailbox can be as simple or sophisticated as it wants about managing its message storage memory, queuing and interrupts, without affecting other units.

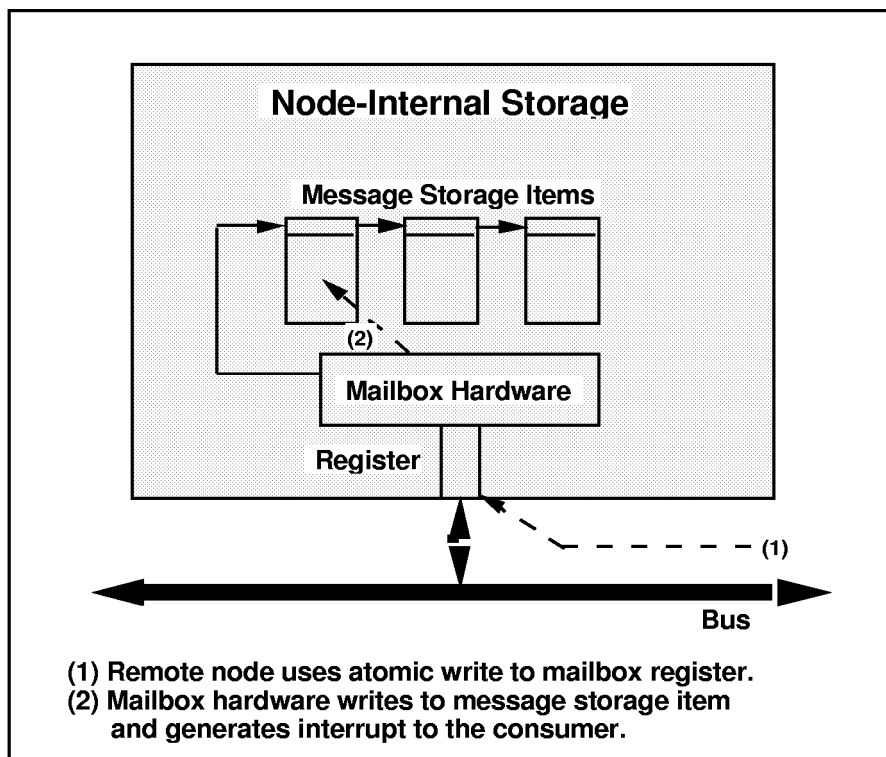


Figure 4.9—Mailbox hardware hides message storage and synchronization

The structures used for the message storage items are only known inside the unit, not to the remote message producers. These may be provided by a FIFO buffer, a linked list (as shown), Circular Queues, etc. These data structures may operate within the unit in a way similar to the DMA schemes described above. But since these details are unit-internal, they are not of concern to other units on the external bus and *are not subject to standardization*. Only the mailbox hardware writes to the message storage items. Producers need only provide hardware to do a writel6, write32, or whatever size atomic write is needed to contain an entire message.^{21, 22} The target unit provides all special mailbox hardware. Mailbox hardware can be very similar to that required for normal dual-port memory. For dual-port memory, the writer gives the address of the RAM cells to be written. For the mailbox, the target unit knows the memory address *a priori*. Bus hardware may absorb data at the full line rate and then transfer the data to RAM. This RAM is probably confined to the same board as the one containing the register. But it could be tied into the cache mechanism and backed by System Memory in another unit.

Note that the message_request and message_response target registers defined in IEEE Std 1212-1991 for each node can be accessed by write and broadcast-write transactions. In the case of broadcast-write transactions, all nodes on the specified bus will receive the transaction. For this reason, broadcast transactions are not useful in the DMA Framework

²¹Notice that the auto-increment of data stream byte locations is done behind the register, rather than on the bus side. All data streams through a single register. In Futurebus+, mailbox registers are placed on 64-byte alignments to possibly simplify write hardware that is cache-aligned, and to allow some of the least significant bits to be used for mask features with broadcast messages.

²²The minimum, usable message size is one pointer wide. This can be used to point to the actual message body, stored in System Memory. Additional bytes are extremely helpful since they allow additional items (like the command and data size) to be known to the consumer for event scheduling without reading System Memory.