INTERNATIONAL STANDARD

ISO/IEC 10857

ANSI/IEEE Std 896.1

First edition
1994-04-27

Information technology – Microprocessor systems – Futurebus+ – Logical protocol specification

Technologies de l'information – Systèmes à microprocesseurs – Futurebus+ – Spécification du protocole logique



Reference number
ISO/IEC 10857 : 1994(E)
ANSI/IEEE
Std 896.1, 1994 Edition

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Abstract: This International Standard provides a set of tools with which to implement a Futurebus+ architecture with performance and cost scalability over time, for multiple generations of single- and multiple-bus multiprocessor systems. Although this specification is principally intended for 64-bit address and data operation, a fully compatible 32-bit subset is provided, along with compatible extensions to support 128- and 256-bit data highways. Allocation of bus bandwidth to competing modules is provided by either a fast centralized arbiter, or a fully distributed, one or two pass, parallel contention arbiter. Bus allocation rules are provided to suit the needs of both real-time (priority based) and fairness (equal opportunity access based) configurations. Transmission of data over the multiplexed address/data highway is governed by one of two intercompatible transmission methods: a) a technology-independent, compelled-protocol, supporting broadcast, broadcall, and transfer intervention (the minimum requirement for all Futurebus+ systems), and b) a configurable transfer-rate, source-synchronized protocol supporting only block transfers and source-synchronized broadcast for systems requiring the highest possible performance. Futurebus+ takes its name from its goal of being capable of the highest possible transfer rate consistent with the technology available at the time modules are designed, while ensuring compatibility with all modules designed to this standard both before and after. The plus sign (+) refers to the extensible nature of the specification, and the hooks provided to allow further evolution to meet unanticipated needs of specific application architectures. It is intended that this International Standard be used as a key component of an approved IEEE Futurebus+ profile.
Keywords: bus architecture, Futurebus+, logical protocol, multiprocessor systems

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 1-55937-373-3

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.

April 27, 1994 *SH16816*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

ISO/IEC 10857 : 1994
[ANSI/IEEE Std 896.1, 1994 Edition]
(Incorporates ANSI/IEEE Std 896.1-1991 and
IEEE Std 896.1a-1993)

Information technology— Microprocessor systems— Futurebus+ — Logical protocol specification

Sponsor

Bus Architecture Standards Committee
of the
IEEE Computer Society



Adopted as an International Standard by the
International Organization for Standardization
and by the
International Electrotechnical Commission



Published by
The Institute of Electrical and Electronics Engineers, Inc.



Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and nongovernmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

In 1993, ANSI/IEEE Std 896.1-1991, together with IEEE Std 896.1a-1993, *Errata, Corrections and Clarifications*, was adopted by ISO/IEC JTC 1, as draft International Standard ISO/IEC DIS 10857. This edition incorporates IEEE Std 896.1a-1993 into the text of ANSI/IEEE Std 896.1-1991.



International Organization for Standardization/International Electrotechnical Commission
Case postale 56 • CH-1211 Genève 20 • Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Introduction

(This introduction is not a normative part of ISO/IEC 10857 : 1994, but is included for information only.)

The following is a list of those who were members of the IEEE Futurebus+ Working Group at the time ANSI/IEEE Std 896.1-1991 was approved:

Paul L. Borrill, *Chair*

Barbara Aichinger
Ray Alderman
Hamid Amirazzi
Duane Anderson
Harrison Beasley
Janos Biri
Martin Blake
Richard Boberg
Andy Bonafini
David Brash
David Brearly
David Brewer
Marc Briel
Charles Brill
Jim Brown
Mark Bunker
John Campbell
Jay Cantrell
Stephen Cecil
Kim Clohessy
Paul Cook
Dante Del-Corso
Ernie Crocker
Jon Crowell
Steve Diess
Paul Dixon
Ian Dobson
Emer Dooley
Sam Duncan
Chris Eck
Bill Evertz
Wayne Fischer
Mike Foster
Joseph George
Larry Gilbert
Jim Goodman
Robert Greiner
David Gustavson
Emil Hahn
David Hartig
David Hawley
Lym Hevle
Billy Ho
Mike Humphrey
John Hyde
Ed Jacques
David James
Greg Jewell
Anatol Kaganovich
Hans Karlsson
David Kemp
Ralph Lachenmaier
Subasis Laha
Cees Lambretche
Dick Lawrence
Mike Lazar
Jim Leahy
Kent Leung
Joel Liblove
Thanos Mentzelopoulos
Klaus Müller
Chris Nichols
Jim Nicholson
Ronald Niederhagen
Mira Pauker
Chet Pawlowski
Clarence Peckham
Shlomo Pri-Tal
Surinder Rai
Mike Raynham
Jack Regula
Bill Ruszczyk
Ali Sarabi
James Scaminaci
Dennis Schmitz
Craig Scott
Don Senzig
Lui Sha
Dan Sieworek
Mike Snodgrass
Michael Sweeney
Fahad Tabrizzi
Matthew Taub
Mike Teener
Judy Teske
Morton Thayer
John Theus
Mike Thompson
Nigel Topham
Mary Vernon
Harvey Walthersdorf
Eike Waltz
Randy Weber
Mike Wenzel
Mike Wiles
Mark Williams
John Wise
David Wright
Dale Younge

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The following persons were on the balloting committee of ANSI/IEEE Std 896.1-1991:

William B. Adams
Sid Ahuja
Mohammad Al-Malki
John Allen
Richard P. Ames
Duane L. Anderson
Jack Arabian
R. V. Balakrishnan
David M. Barnum
Harrison A. Beasley
Janos Biri
Kyle M. Black
John Black
William P. Blase
Jack L. Blevins
David Brearley
Charles Brill
Lyle Burnett
Luis-Felipe Cabrera
Clyde Camp
Donald Chi
Kim Clohessy
David Cohen
Paul D. Cook
Robert Crowder
Jonathan C. Crowell
Philip D'Angelo
Ana Maria Dealvare
Stephen Deiss
Dante Del Corso
Su Dongzhuang
Mike Dorsett
Samuel H. Duncan
Sourav Dutta
Jeffrey S. Ebeling
William P. Evertz
Harry D. Feit
Wayne Fischer
Gordon Force
Andrew Fraser
Joseph D. George
Andy Glew
Patrick Gonia
Willard Graves
William Groseclose
David B. Gustavson
Thomas W. Harkaway
David Hawley
Herbert Hecht
Rick Henderson
Frank Hom
Scott Hopkinson
Zoltan R. Hunor
Peter J. Ilieve
Bob Jacobsen
Edgar Jacques
David V. James
Kenneth Jansen
Jack R. Johnson
Anatol Kaganovich
Hans Karlsson
David Keeney
Willis K. King
Hubert Kirrman
Ernst H. Kristiansen
Thomas M. Kurihara
Tuvia Lamdan
Glen Langdon
Thomas Leonard
Per Lindman
William Lindow
Rollins Linser
Wayne M. Loucks
Anthony G. Lubowe
Andy J. Luque
Roy Maurer
William McDonald
Darrell B. McIndoe
Bruce Millard
Lee Minsuk
James M. Moidel
James Moloney
J.D. Nicoud
Tadahiko Nishimukai
Duane J. Northcutt
Gregory C. Novak
Michael Orlovsky
Jame R. Otto
Dick Palmer
Mira Pauker
Donald Pavlovich
James M. Pexa
Arthur V. Pohm
Bruno R. Preiss
Shlomo Pri-tal
Greg Prom
Richard Rawson
Michael Raynham
Ed Rodriquez
Tom Sakoda
Debabrata Sarma
Carl Schmiedekamp
Norman Schneidewind
Eugene C. Schramm
David Seraphin
Philip Shutt
Michael R. Sitzer
Michael Smolin
Benjamin Stoppe, Jr.
Paul Sweazey
Daniel Tabak
Darius Tanksalvala
Daniel Tarrant
Michael Teener
Michael G. Thompson
Carsten Thomsen
Joseph P. Trainor
Robert Tripi
Joseph G. Tront
Robert J. Voight
Eike Waltz
David R. Weller
Walter L. Whipple
Thomas Wicklund
Hans A. Wiggers
Mark Williams
John S. Willy
Andrew Wilson
John Wise
Joel Witt
David L. Wright
Qiufeng Wu
Oren Yuen

When the IEEE Standards Board approved ANSI/IEEE Std 896.1-1991 on September 26, 1991, it had the following membership:

Marco Migliaro, *Chair*

Donald C. Loughry, *Vice Chair*

Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Clyde Camp
James M. Daly
Donald C. Fleckenstein
Jay Forster*
David F. Franklin
Ingrid Fromm
Thomas L. Hannan
Donald N. Heirman
Kenneth D. Hendrix
John W. Horch
Ben C. Johnson
Ivor N. Knight
Joseph L. Koepfinger*
Irving Kolodny
Michael A. Lawler
John E. May, Jr.
Lawrence V. McCall
Donald T. Michael*
Stig L. Nilsson
John L. Rankine
Ronald H. Reimer
Gary S. Robinson
Terrance R. Whittemore

*Member Emeritus

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The following is a list of those who were members of the IEEE Futurebus+ Working Group at the time IEEE Std 896.1a-1993 was approved:

Samuel H. Duncan, *Chair*

Harrison Beasley
Kim Burris
Jay Cantrell
Steve Cecil
Steve DiCamillo
R. Paul Dixon
Ian Dobson
Karl Franklin
Joseph D. George
Claes-Goran Gustavsson
Emil Hahn
Peter Izzo
Ed Jacques
Greg Jewell
Jim Leahy
Jeff Lear
Thanos Mentzelopoulos
Michael Munroe
Robert Schetlick
Gene Schramm
Richard Spratt
John Theus
Dean Van De Walker
Robert Widlicka

The following persons were on the balloting committee of IEEE Std 896.1a-1993:

Edward W. Aichinger
Ray S. Alderman
Richard P. Ames
Keith D. Anthony
Harrison A. Beasley
John Black
Charles Brill
Andrew J. Brough
Clyde Camp
Stephen J. Cecil
Andy Cheese
Kim Clohessy
Steven Cobb
David Cohen
Steven R. Corbesero
Ian Dobson
Jean-Jacques Dumont
Samuel Duncan
Christopher Eck
Wilhelm P. Evertz
Wayne Fischer
Gordon Force
Paul Fulton
Julio Gonzalez-Sanz
John Griffith
Michael C. Hayward
Edgar Jacques
Ralph Lachenmaier
Lak Ming Lam
Michael Lambrou
Karl E. McClure
Thanos Mentzelopoulos
Bruce Millard
Brian D. Morrison
Klaus Dieter Mueller
Elwood Parsons
Chandresh J. Patel
Steve Quinton
Michael L. Roby
Frederick E. Sauer
Robert Schetlick
Don Denzig
Patricia Smith
Joanne Spiller
Richard Spratt
Michael G. Thompson
Joseph P. Trainor
Robert Tripi
Yoshiaki Wakimura
Eike Waltz
Dave Wickliff
Robert Widlicka
Joel Witt
Mark Woodbury
David L. Wright
Yoshio Yamaguchi

When the IEEE Standards Board approved IEEE Std 896.1a-1993 on September 15, 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

IEEE Std 896.1-1991 was approved by the American National Standards Institute on April 28, 1992.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Contents

CLAUSE PAGE

1. Overview 1

1.1 Scope 1
1.2 Normative references 3

2. Definitions and structure 4

2.1 Special word usage 4
2.2 Definitions 4
2.3 Signal conventions 8
2.4 Document structure 9
2.5 Futurebus+ logo 10
2.6 Bus line description 11
2.7 Attribute cross reference 14
2.8 Implementation mnemonics 24

3. Bus signaling environment 26

3.1 Description 26
3.2 Specification 26

4. Centralized arbitration 28

4.1 Description 28
4.2 Specification 30

5. Distributed arbitration and arbitrated messages 31

5.1 Description 31
5.2 Specification 46

6. Parallel protocol 57

6.1 Description 57
6.2 Specification 96

7. Bus/system management 128

7.1 Description 128
7.2 Specification 136

8. Cache coherence 144

8.1 Description 144
8.2 Specification 168

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

CLAUSE PAGE

9. Message passing 174

9.1 Description 174
9.2 Specification 190

ANNEX

Annex A Bibliography 200

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Information technology—Microprocessor systems—Futurebus+ — Logical protocol specification

1. Overview

1.1 Scope

This International Standard specifies the logical (relative timing and behavioral protocol) layer for a set of signal lines that constitute a multiple segment bus architecture, and for the interfacing of modules connected to a bus segment. This International Standard is intended to be used as a component within a profile (a collection of related specifications that must be used together by a product in order to claim conformance to a standard) to build systems with higher levels of compatibility.

Futurebus+ provides the means for the transfer of binary information between boards over one or more logical buses. Boards may contain any combination of one or more processors and local resources such as cache, memory, peripheral and communication controllers, etc. Figure 1 shows a block diagram of a typical application of Futurebus+.

Protocols are specified for the allocation of bus time to modules that need to conduct transactions with other modules over the bus. However, this International Standard does not mandate the priority rules for modules to use when competing for use of the bus. These are considered the privilege and responsibility of the system integrator. The International Standard includes a complete set of signaling rules to be followed by all modules in both the distributed and centralized control acquisition processes leading to bus mastership (clauses 4 and 5). The International Standard also gives a comprehensive set of signaling rules for all modules participating in a bus transaction (clause 6).

Most of the transfer protocols in this International Standard are *compelled*; that is, they are governed by a pure cause-and-effect relationship. This is what gives this International Standard its technology-independent nature. The compelled signaling provides a designer with a logical simplicity for what takes place in the protocols. As a result, there will be maximum compatibility between products designed to this International Standard throughout its operational lifetime.

With any bus, there is the dilemma of how much the standard should specify. There must be a balance between ensuring that all boards designed by a variety of manufacturers can operate together, while not restricting the users of the bus to any preconceived system design. Although the scope of this International Standard has been restricted to exclude many of the system requirements associated with bus-based computer systems, these are being addressed in companion standards.

The common control and register interface to this series of standards for the Futurebus+, and to other proposed IEEE standards (in particular, IEEE Std 1596-1992 [B12][1], IEEE P1014.1 [B2], and IEEE

[1] The numbers in brackets correspond to those of the bibliography in annex A.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

P1394 [B11]), is embodied in the unified CSR architecture standard, IEEE Std 1212-1991 [B7], along with a unified DMA architecture for moving data around a system without the need to pass through a processor (IEEE Std 1212.1-1993 [B8]).

This set of protocols has been designed to be as close to technology-independent as possible while maintaining a very high level of efficiency and performance. The bus signals may be implemented using any technology (TTL, Backplane Transceiver Logic, ECL, CMOS, GaAs, etc.) so long as the Futurebus+ signaling conditions are met (incident wave switching on the transmission-line signaling environment, along with the constraints on skew, crosstalk, and transmission reliability). However, in the interest of maximum compatibility between product families, implementations are expected to be associated with one or more IEEE Futurebus+ profiles, which specify the physical layer and set of transactions to suit a particular family of applications.

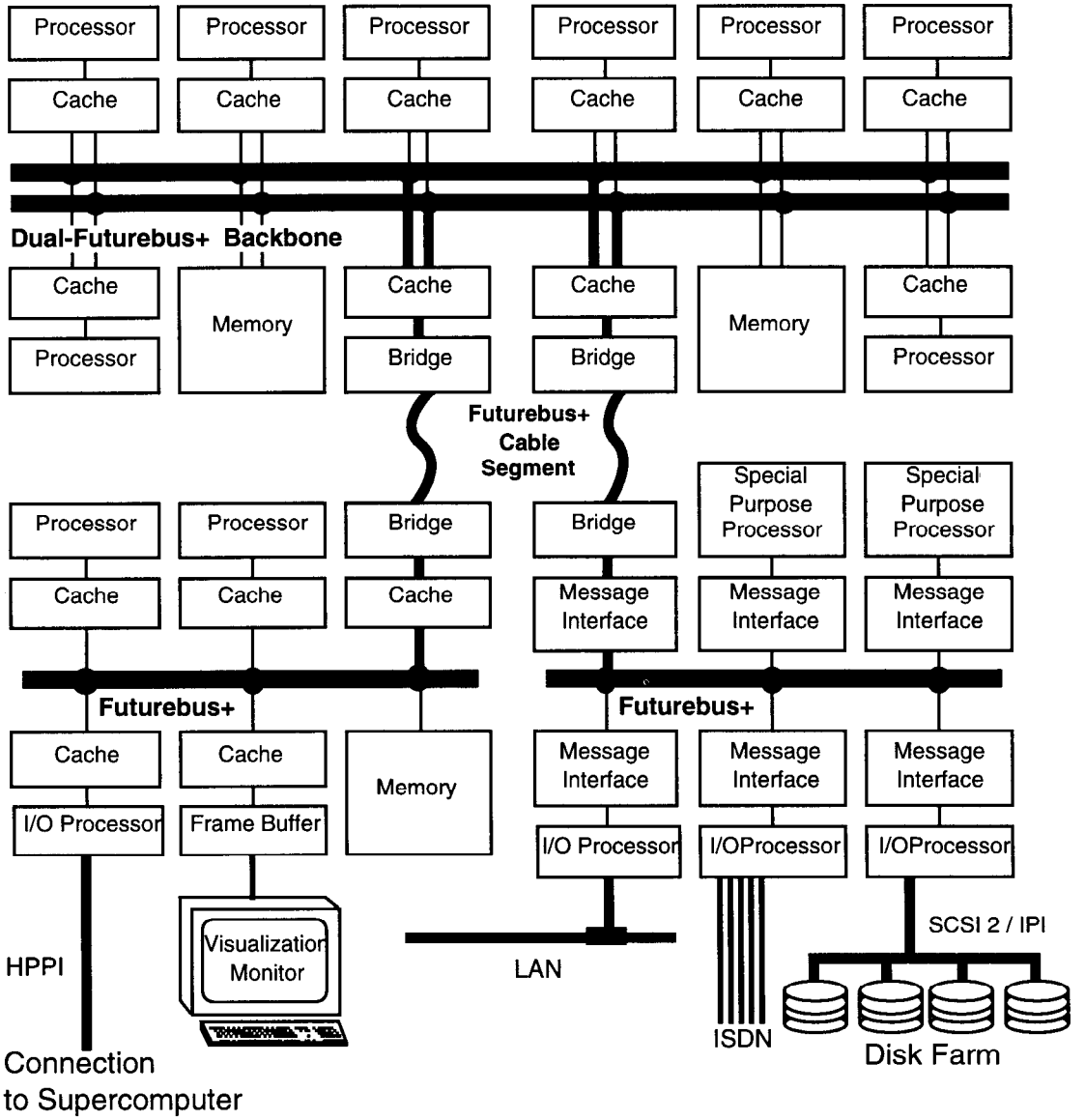


Figure 1—Interfaces in a family of typical Futurebus+ systems

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.2 Normative references

The following standards contain provisions which, through references in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent edition of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

IEEE Std 896.2-1991, IEEE Standard for Futurebus+ — Physical Layer and Profile Specifications.[2]

IEEE Std 896.3-1993, IEEE Recommended Practices for Futurebus+.[3]

[2] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[3] As this standard goes to press, IEEE Std 896.3-1993 is not yet published. It is, however, available in manuscript form from IEEE. Anticipated publication date is May 1994.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2. Definitions and structure

2.1 Special word usage

2.1.1 may: A keyword indicating flexibility of choice with no implied preference.

2.1.2 shall: A keyword indicating a mandatory requirement. Designers must implement all such mandatory requirements to ensure interoperability of ISO/IEC 10857 conformant products and claim conformance to this International Standard.

2.1.3 should: A keyword indicating flexibility of choice with a strongly preferred implementation. The phrase *it is recommended* is used interchangeably with the keyword *should.*

2.2 Definitions

2.2.1 activate: a) The action of applying a set of signals to a group of bus lines. b) The state of a group of bus lines when they carry signals.

2.2.2 address-only transaction: A bus transaction that does not include a data phase. The only information transferred is contained within the connection phase and, in some cases, the disconnection phase.

2.2.3 arbitration: The process of selecting the next bus master.

2.2.4 arbitrated message: A number broadcast on the arbitrated message bus lines to all modules on the bus.

2.2.5 assert: a) The action of applying a logic one signal to a bus line. b) The state of a bus line when the signal it carries represents a logic one.

2.2.6 * (asterisk): When appended to a signal's name, the suffix "*" indicates that the logic one state of the signal is such that it will override the logic zero state applied by any other module on that line.

2.2.7 beat: An event that begins with the transition on a synchronization line by the master, followed by the release of an acknowledge line by one or more slaves. Command and data information may be transferred from the master to one or more slaves in the first half of the beat. During the second half of the beat the slaves may transfer capability, status, and data information back to the master.

2.2.8 block copy: A block copy operation is characterized by a long series of read or write transactions to sequential memory locations.

2.2.9 bus bridge: A bus bridge is an interconnect between two or more buses that provides signal and protocol translation from one bus to another. The buses may adhere to different bus standards for mechanical, electrical, and logical operation (such as a bus bridge from Futurebus+ to VMEbus or to MULTIBUS II).

2.2.10 bus line: The medium for the transmission of signals. Since Futurebus+ requires drivers with wire-OR capability, a bus line may be driven by several modules simultaneously. Therefore, the signal carried by the bus line is the combination of signals applied to that line from each module.

2.2.11 bus tenure: The duration of a master's control of the bus; i.e., the time during which a module has the right to initiate and execute bus transactions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.2.12 bus transaction: An event initiated with a connection phase and terminated with a disconnection phase. Data may or may not be transferred during a bus transaction. *See:* **transaction.**

2.2.13 busy: If a slave is unable to accept a bus transaction from a master, it may issue a busy status to the master of the transaction. The master must relinquish the bus and may reacquire the bus and retry the transaction after a suitable time interval.

2.2.14 byte: A set of eight adjacent binary digits.

2.2.15 byte lane: A data path formed by eight data lines and one parity line and used to carry a single byte between system modules.

2.2.16 cache coherence: A system of caches is said to be coherent with respect to a cache line if each cache and main memory in the coherence domain observes all modifications of that same cache line. A modification is said to be observed by a cache when any subsequent read would return the newly written value.

2.2.17 cache memory: A buffer memory inserted between one or more processors and the bus, used to hold currently active copies of blocks from main memory. Cache memories exploit spatial locality by what is brought into a cache. Temporal locality is exploited by the strategy employed for determining what is removed from the cache.

2.2.18 coherence domain: A region in a multiple-cache system, inside of which, cache consistency measures are enforced. In a system that contains bus bridges, a coherence domain may or may not be extensible beyond the local bus through a bus bridge to remote buses.

2.2.19 coherence line: A data block for which cache consistency attributes are maintained.

2.2.20 compelled data transfer protocol: A technology-independent transfer mechanism in which the slave is compelled to provide a response before the master proceeds to the next transfer.

2.2.21 competitor: A module actively participating in the current control acquisition cycle of the arbitration process.

2.2.22 connected transaction: A transaction in which both the request and response are performed within the same bus transaction.

2.2.23 connection phase: A beat that begins with the assertion of the address synchronization line followed by the release of an address acknowledge line. It is used to broadcast the address and command information. Modules determine whether they wish to take part in the transaction based on this information.

2.2.24 control acquisition: The total of all bus activity associated with acquiring exclusive control of the bus.

2.2.25 copyback cache: A cache memory scheme with the attribute that data written from the processor is normally written to the cache rather than the main memory. Modified data in the cache is written to the main memory to avoid loss of the data when a cache line flush or replacement occurs.

2.2.26 CSR: Control and status register.

2.2.27 CSRA: Control and status register architecture. (See IEEE Std 1212-1991 [B7].)

2.2.28 data phase: A period within a transaction used to transfer data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.2.29 deadlock: A state that occurs when modules are awaiting actions that can only be performed by those waiting, and those waiting cannot perform the actions.

2.2.30 disconnection phase: A period within a transaction used to return the bus signals to their quiescent state. In addition, this phase might be used to transfer additional information required to perform or abort the requested operation.

2.2.31 doublet: A set of two adjacent bytes.

2.2.32 entrant: A live inserted module in the process of aligning itself with the arbitration protocol.

2.2.33 forward progress: A state in which a module is not blocked from performing the tasks necessary to achieve its goal. Forward progress is guaranteed only in the absence of deadlock or starvation.

2.2.34 geographical address: A unique identifier assigned to each physical module slot on the bus and assumed by any module connected to that slot.

2.2.35 global identification: A unique identifier assigned to each physical module slot in a system. This identifier would typically include both a bus identifier and a slot identifier. IEEE Std 1212-1991 [B7] specifies the format for such a global identifier.

2.2.36 intervening slave: The participating slave that, although not the repository of last resort of the requested data, finds it necessary to prevent the repository of last resort from providing the requested data. Having done so, the intervening slave provides the data instead.

2.2.37 livelock: A metastable situation in which some modules acquire and release resources in such a way that none of them makes forward progress.

2.2.38 locking: A facility whereby a module is requested to guarantee exclusive access to addressed data, blocking other modules from accessing that data. This allows indivisible operations to be performed on addressed resources.

2.2.39 master: A module that has acquired control of the bus through the control acquisition procedure.

2.2.40 master elect: A module that has won the most recent arbitration competition.

2.2.41 module: A collection of circuitry designed to perform specific functions that include an interface to Futurebus+.

2.2.42 monarch processor: The processor selected to manage the configuration and initialization of all modules on one logical bus. *Also:* monarch. An *emperor processor* is the monarch processor selected to direct the configuration and initialization of an entire system with multiple interconnected logical buses.

2.2.43 octlet: A set of eight adjacent bytes.

2.2.44 packet data transfer protocol: A very fast but technology-dependent noncompelled transfer mechanism that uses a compelled protocol over the entire packet to provide flow control.

2.2.45 parallel contention arbitration: A process whereby modules assert their unique arbitration number on a parallel bus and release signals according to an algorithm such that after a period of time the winner's number appears on the bus.

2.2.46 participating slave: A slave involved in a transaction as a selected slave, an intervening slave, a broadcast slave, or a slave involved in multiple packet mode.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.2.47 preemption: The release of the bus by the current bus master due to the request of another module.

NOTE—In some systems, any module may cause preemption; in others, only a module with a higher priority request may cause preemption.

2.2.48 quadlet: A set of four adjacent bytes.

2.2.49 release: a) The action of applying a logic zero signal to a bus line. b) The state of a bus line when the signal it carries represents a logic zero.

2.2.50 repository of last resort: In a hierarchical memory (or cache-based) environment, a storage location that "owns" the only, or last remaining, copy of sharable data.

NOTE—It may be a unique source, an ultimate destination, or simply a "safe" repository of data that may not be invalidated, unless action is taken to preserve a copy of that data at some higher level in the memory (or cache) hierarchy. In a cache-only Futurebus+ system (e.g., one where even the main DRAM storage is also designed as a hardware cache), the repository of last resort begins life as the binding of an address to a physical location in one of the caches, along with the creation of the data by initialization, a copy from some higher level in the memory hierarchy, or by its arrival from some I/O device. This data may migrate around the system, and be owned by different caches at different times, provided no less than one copy of that data is maintained somewhere. A repository of last resort may end its life by an explicit instruction to "destroy" the data by migration to a higher level in the memory (or cache hierarchy), or by transfer of ownership through some I/O device to another system, storage device, or display.

2.2.51 request: A command, generated by a requester, to initiate an action on a responder. For a processor-to-memory read transaction, for example, the request transfers the memory address and command from the processor to memory. In the case of a split transaction, the request would be a separate bus transaction. In the case of a connected transaction, the request would be the connection phase of a bus transaction.

2.2.52 requester: A module that initiates a transaction by sending a request (containing address, command, and sometimes data).

2.2.53 responder: A module that completes a transaction by sending a response (containing the completion status and sometimes data).

2.2.54 response: A reply, generated by a responder, to complete a transaction initiated by a requester. For a processor-to-memory read transaction, for example, the response returns the data and status from the memory to the processor. In the case of a split transaction, the response would be a separate bus transaction. In the case of a connected transaction, the response would be the data and disconnection phases of a bus transaction.

2.2.55 round robin: A bus allocation rule where, after a module acquires and uses the bus, it will not be granted use of the bus again until all other modules currently requesting the bus at the same priority level have had control of the bus.

2.2.56 selected slave: A slave is selected during the connection phase by the master when it recognizes its address on the bus lines.

2.2.57 shared memory: The address space in the system accessible to all caching modules.

2.2.58 signal names: Where a group of bus lines are represented by the same letters, the lines within the group are numbered; e.g., **AD0***, **AD1***, **AD2***, etc. In order to represent a group of lines or signals in more convenient form, notation such as **AD[63..0]*** is used. Also, in these examples, the notation **AD[ ]*** is used to refer to all of the lines within the group.

2.2.59 skew: The difference between the propagation delays of two or more signals on any bus line.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.2.60 slave: A module that can be addressed and is capable of participating in bus transactions.

2.2.61 snarf: The action taken by a a module when it takes a copy of data passing by on the bus, even though it did not request it.

2.2.62 snoop: The action taken by a module on a transaction when it is not the master that originated the transaction or the repository of last resort for the data, but it still monitors the transaction. Cache memories snoop transactions to maintain coherence.

2.2.63 spatial locality: The tendency for a program to reference closely related clusters of memory addresses over short time intervals.

2.2.64 split transaction: A system transaction in which the request is transmitted in one bus transaction and the response is transmitted in a separate subsequent bus transaction.

2.2.65 starvation: A system condition that occurs when one or more module(s) perform(s) no useful work for an indefinite period of time due to lack of access to the bus or other system resources.

2.2.66 strong sequential consistency: A state exhibited by a system when each participating cache in the system observes all modifications to lines within itself in the same order as all other participating caches in the system. *See also:* **weak sequential consistency.**

NOTE—Futurebus+ allows modules to issue transactions that dynamically choose between following a weakly consistent behavior model (which implies greater concurrency and hence higher performance) and strongly consistent behavior models (which may be necessary to ensure correct operation of an algorithm written by a programmer not cognizant of the concurrency possible in different parts of a system).

2.2.67 system transaction: A complete operation, such as a memory read or write, as viewed from the initiating unit. A system transaction can be translated into one or more bus transactions by the Futurebus+ interface to complete the operation.

2.2.68 temporal locality: The tendency for a program to reference the same memory locations over short time intervals.

2.2.69 transaction: An event initiated with a connection phase and terminated with a disconnection phase. Data may or may not be transferred during a transaction. Often used instead of the more precise phrase "bus transaction" for the sake of brevity. *See:* **bus transaction; system transaction.**

2.2.70 unselected slave: A slave that does not recognize its address on the bus lines during the connection phase of a bus transaction.

2.2.71 weak sequential consistency: A state exhibited by a system when references to global synchronizing variables exhibit strong sequential consistency, and if no reference to a synchronizing variable is issued by any processor until all previous modifications to global data have been observed by all caches, and if no reference to global data is issued by any processor until all previous modifications to synchronizing variables have been observed by all caches. *See also:* **strong sequential consistency.**

2.3 Signal conventions

Figure 2 shows an example of the signal conventions used in this International Standard. The example is shown with open collector technology (although this International Standard is independent of this logic family). The signal that a specific module applies to the input of its driver is designated by a lowercase label; e.g., **ai**. The signal that a specific module applies to a bus line is designated by a lowercase label with an

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

asterisk; e.g., **ai***. The signal that appears on a bus line as the result of the combined signals applied to it by all modules is designated by an uppercase label with an asterisk; e.g., **AI***.

When appended to a signal name, the suffix "**f**" (filtered) refers to the bus line signal after it has passed through a receiver and a wire-OR glitch filter (integrator). For example, **AIf** refers to the signal on the **AI*** line, after it has passed through an inverting receiver to become **AI** and a wire-OR glitch filter to become **AIf**.

In a label with an appended asterisk (e.g., **WR***), a "0" would mean the signal was released. A "1" would mean an asserted signal.

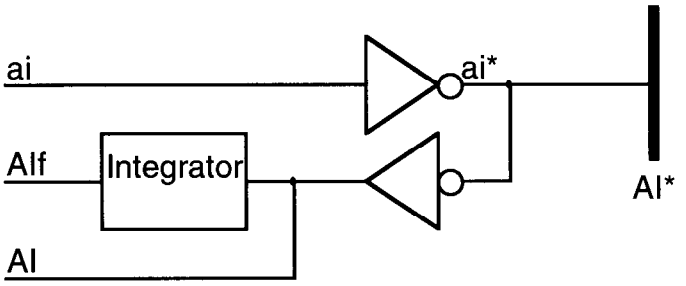


Figure 2—Signal conventions

2.4 Document structure

Each of clauses 3 through 9 is divided into two subclauses: one containing descriptions and one containing specifications. The numbers of the description subclauses begin with the clause number followed by ".1". The numbers of the specification subclauses begin with the clause number followed by ".2". For example, the description subclause of clause 3 is numbered 3.1, and the specification subclause is numbered 3.2.

The description subclauses are intended to help designers understand the operation of the bus and are *not intended to imply any requirements on equipment* built to this International Standard. The specification subclauses contain *all the requirements* with which equipment designed to this International Standard shall comply.

The specification subclauses are written using attributes (including CSR bits), states, and equations. This format, although somewhat difficult to read, allows the specification to be precise. Designers may wish to read all of the description subclauses prior to reading the specification subclauses.

Attributes are identified by uppercase with the underscore character separating words. For example, COMPELLED_STATUS, MASTER, and MASTER_ELECT are attributes. The symbols used in equations in order of precedence are as follows:

- ~ Negation
- & Logical AND
- | Logical OR

For example, the specification "A COMPETITOR shall assert **ab6*** when **cn6** & COMPETE & (**cn7** | ~**AB7***)" would mean: A module with the COMPETITOR attribute set shall assert the signal **ab6*** when the **cn6** signal is set and the COMPETE attribute is set and either **cn7** is set or **AB7*** is released.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Attributes are logical entities that do not necessarily correspond to circuitry on a module. A machine's internal state variables may or may not conform to the attributes described in this International Standard. Any machine whose external behavior is equivalent to the specified machine conforms to this International Standard.

Although this International Standard indicates, where appropriate, possible areas where a designer may encounter metastability (the misbehavior of digital logic at the boundary between two synchronization domains), no guarantee is given that those warnings cover all possible causes of metastability; avoiding metastability must ultimately be the responsibility of the designer, since different application architectures require different synchronization boundaries.

2.5 Futurebus+ logo

The IEEE 896 logo shown in figure 3 is copyrighted by the IEEE Computer Society and may be used only on products that fully conform to all mandatory requirements of an IEEE-approved 896 profile document, or adhere to either this document (ISO/IEC 10857 : 1993) or to earlier editions, IEEE Std 896.1-1991 or IEEE Std 896.1-1987.



Figure 3—IEEE 896 logo

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.6 Bus line description

Table 1 shows the list of bus lines, while 2.6.1.1–2.6.1.9 give a brief description of their function. The signals that use these lines are more fully described in later clauses.

Table 1—List of bus lines

Data Path Width		32-bit	64-bit	128-bit	256-bit
Information lines					
AD[63..0]*	Address/Data	32	64	64	64
D[255..64]*	Data	—	—	64	192
BP[31..0]*	Bus Parity	4	8	16	32
TG[7..0]*	TaG	8	8	8	8
TP*	Tag Parity	1	1	1	1
CM[7..0]*	CoMmand	8	8	8	8
CP*	Command Parity	1	1	1	1
ST[7..0]*	STatus	8	8	8	8
CA[2..0]*	CApability	3	3	3	3
Synchronization lines					
AS*, AK*, AI*	Address handshake	3	3	3	3
DS*, DK*, DI*	Data handshake	3	3	3	3
ET*	End of Tenure	1	1	1	1
Arbitrated message lines					
AB[7..0]*	Arbitrated Message Bus	8	8	8	8
ABP*	Arbitrated Message Bus Parity	1	1	1	1
AP*, AQ*, AR*	Arbitrated Message Synchronization	3	3	3	3
AC[1..0]*	Arbitrated Message Condition	2	2	2	2
Reset/bus initialize line					
RE*	REset	1	1	1	1
Central arbitration					
RQ[1..0]*	ReQuest	2	2	2	2
GR*	GRant	1	1	1	1
PE*	PreEmption	1	1	1	1
GA[4..0]*	Geographical Address	5	5	5	5
Total active signals		96	132	204	348[§]

[§]NOTE—For the number of 0V logic lines, serial bus lines, and other profile-related information, refer to IEEE Std 896.2-1991.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.6.1 Information lines

The following bus lines are used during the execution of bus transactions.

2.6.1.1 AD[63..0]* address/data/lane deselect/request address/return address

These multiplexed bidirectional address/data lines carry the address during an address transfer. The address is always driven by the master and received by one or more slaves. During the data transfer these lines carry data. Data is driven by the master during write transactions, and by the selected or intervening slave during read transactions. During the first data beat of partial transfers **AD[31..0]*** carry lane deselect signals. During cached split transactions **AD[31..0]*** carry the requesting module's address and priority and the responding module's status. During noncached split transactions **AD[31..0]*** carry the requesting module's global identification and transaction identification.

2.6.1.2 D[255..64]* data lines

These bidirectional data lines carry the data during a data transfer. Data is driven by the master during write transactions and the response portion of split read transactions. Data is driven by the selected or intervening slave during read transactions.

2.6.1.3 CM[7..0]* command

A set of lines that carry command information from the master to one or more slaves for the current transaction.

2.6.1.4 CP* command parity

The line that carries the parity for the command lines.

2.6.1.5 ST[7..0]* status

A set of lines that are activated by slaves to respond to the master's command. **ST[7..0]*** provide information both about the status of slaves and their disposition in regards to the requested transfer.

2.6.1.6 CA[2..0]* capability

A set of lines that a module activates to declare its ability to accommodate major modes of bus transactions. **CA[2..0]*** provide the basic mechanism that allows modules with various transaction capabilities to coexist in the system; e.g., a module that can respond to packet data transfers and a module that can only execute compelled data transfers.

2.6.1.7 BP[31..0]* bus parity

These lines carry parity for the address/data lines. There is one parity line for every 8 address/data lines.

2.6.1.8 TG[7..0]* tag

These bidirectional lines carry additional information related to the address/data lines. This International Standard does not prescribe the use of **TG[7..0]***; the system designer may use them to fit the specific requirements of the application. The number of tag lines active is specified in a control and status register.

2.6.1.9 TP* tag parity

The line that carries the parity code for **TG[7..0]***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.6.2 Synchronization lines

The synchronization lines coordinate the exchange of the address, command, capability, status, and data during a transaction.

2.6.2.1 AS* address synchronization

A synchronization signal that is asserted during the connection phase by the master to inform the system's slaves that the address and command information are valid. **AS*** is released during the disconnection phase to indicate that the disconnection data and command information are valid.

2.6.2.2 AK* address acknowledge

An acknowledge signal released by all modules during the disconnection phase to indicate to the master that their status information is valid.

2.6.2.3 AI* address acknowledge inverse

An acknowledge signal released by all modules during the connection phase to indicate to the master that their status and capability information is valid.

2.6.2.4 DS* data synchronization

A synchronization signal asserted or released by the master during the data transfer phase to indicate to the participating slaves that command information and write data are valid or that the master is ready to receive read data.

2.6.2.5 DK* data acknowledge

An acknowledge signal released by the participating slaves during the even beats or even packets of the data phase to indicate to the master that status information and read data are valid or write data has been received.

2.6.2.6 DI* data acknowledge inverse

An acknowledge signal released by the participating slaves during odd beats or odd packets of the data phase to indicate to the master that status information and read data are valid or write data has been received.

2.6.2.7 ET* end of tenure

This signal marks the end of tenure of the current master. The release of this signal notifies the master elect that it is the new master.

2.6.3 Distributed arbitration and arbitrated message lines

2.6.3.1 AB[7..0]* arbitration bus

A set of lines that carry a number that signifies the precedence of competitors during the arbitrated message process and the distributed arbitration process.

2.6.3.2 ABP* arbitration parity

ABP* carries the parity for **AB[7..0]***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.6.3.3 AP*, AQ*, AR* control acquisition synchronization

Handshake lines that perform a cyclic handshake sequence that controls the sequencing of the arbitrated message process and the distributed arbitration process.

2.6.3.4 AC[1..0]* arbitration condition

Lines that control the arbitrated message process and the distributed arbitration process.

2.6.4 RE* reset/bus initialize

A line asserted by a module to initialize the bus interface logic of all modules to a predefined state, and provide a reset signal for use on the boards. The duration of the asserted state on **RE*** is used to decide whether the signal represents live insertion, bus initialize, or system reset.

2.6.5 Central arbiter

In systems implementing a central arbiter, these signals are used for bus master selection as described in the following subclauses.

2.6.5.1 PE* preemption

The central arbiter asserts this line to indicate to the current master that a preemptive condition exists and the master should relinquish bus mastership.

2.6.5.2 GR* grant

This line is asserted by a central arbiter to grant bus mastership. There is a separate grant line from the central arbiter to each module on the bus.

2.6.5.3 RQ[1..0]* request

One or both of these lines are asserted by modules to request bus mastership. There are two separate request lines from each module on the bus to the central arbiter.

2.6.6 GA[4..0]* geographical address

A set of static signals from the backplane that encode the slot number of the module.

2.7 Attribute cross reference

Each attribute lists the clauses that refer to it. A clause in **boldface type** includes the definition of the attribute.

32_BIT_ADDRESS_CAPABLE **7.2.4.1.1**

32_BIT_CAPABLE **6.2.1.3**

32_BIT_DATA_CAPABLE **7.2.4.1.1**

64_BIT_ADDR **6.2.1.3**

64_BIT_DATA **6.2.1.3**, 6.2.2.5.1, 6.2.2.5.3

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

64BIT_ADDRESS_CAPABLE 6.2.1.3, **7.2.4.1.1**

64BIT_DATA_CAPABLE **7.2.4.1.1**

128_BIT_DATA **6.2.1.3**, 6.2.2.5.1, 6.2.2.5.2, 6.2.2.5.3

128BIT_DATA_CAPABLE **7.2.4.1.1**

256_BIT_DATA **6.2.1.3,** 6.2.2.5.1, 6.2.2.5.2, 6.2.2.5.3

256BIT_DATA_CAPABLE **7.2.4.1.1**

ADDR_DECODED 6.2.1.7.2, **6.2.1.8,** 6.2.2.1.3, 6.2.3.2, 8.2.2, 9.2.1

ADDR_WIDTH **6.2.1.3**, 6.2.2.2.8, 6.2.2.5.1, 6.2.2.5.3

ADDRESS_ONLY **6.2.1.4,** 8.2.3

ADDRESS_ONLY_LOCKED **6.2.1.4,** 6.2.1.5, 6.2.1.8, 6.2.2.5.1

ADDRESS_ONLY_UNLOCKED **6.2.1.4,** 6.2.1.8, 6.2.2.5.1

ADDRESS_PARITY_ERROR **6.2.1.7.2**, 6.2.1.8

ADDRESS_TAG_PARITY_ERROR **6.2.1.7.2**

ARBITRATED_MESSAGE_REQUEST **5.2.1**, 5.2.4, 5.2.7.1.1, 5.2.9.1, 5.2.9.2, 5.2.9.6, 7.2.4.2.1

ARBITRATED_MESSAGE_SENT **5.2.1**, 5.2.9.1, 5.2.9.2, 5.2.9.6

ARBITRATION_COMPARISON_ERROR **5.2.6**

ARBITRATION_ERROR 5.2.4, **5.2.6**, 5.2.7.2.1, 5.2.8.1, 5.2.8.3–5.2.8.5, 5.2.9.1, 5.2.9.3–5.2.9.5

ARBITRATION_OWNER **5.2.3**, 5.2.4, 5.2.7.1.3, 5.2.8.1, 5.2.8.2, 5.2.8.5, 5.2.8.6

ARBITRATION_PARITY_ERROR **5.2.6**, 5.2.8.1, 5.2.8.4, 5.2.9.1, 5.2.9.4

ARBITRATION_TIMEOUT **5.2.6**, 5.2.7.1.2, 5.2.7.1.3, 5.2.8.1, 5.2.8.3, 5.2.8.5, 5.2.9.1, 5.2.9.3, 5.2.9.5

ASSERT_CS 6.2.2.4.5, **8.2.2**

ATTRIBUTES_QUEUED **8.2.2**

BEAT_ERROR **6.2.1.7.2**, 6.2.2.4.7

BEAT_ERROR_STATUS **6.2.1.7.1**, 6.2.1.8

BROADCALL_ADDRESS **6.2.1.7.2**, 6.2.2.4.4

BROADCAST **6.2.1.7.2,** 6.2.1.8, 6.2.2.1.5, 6.2.1.8, 6.2.2.4.4, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.14, 9.2.1

BROADCAST_SELECT_ATTRIBUTE **9.2.1**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

BROADCAST_STATUS **6.2.1.7.1,** 6.2.1.8, 6.2.2.1.4, 6.2.2.1.5, 6.2.2.1.6, 6.2.2.5.1–6.2.2.5.3, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.22

BUS_HOLD 5.2.1–5.2.3, 6.2.1.8, 6.2.2.1.1, **7.2.1**

BUS_IDLE **7.2.1**

BUS_IDLE_1US 5.2.7.1.3, 6.2.2.1.3, **7.2.1**, 7.2.2, 7.2.3.4

BUS_INIT **7.2.1**, 7.2.3.3, 7.2.4.2.1–7.2.4.2.6

BUSY **6.2.1.7.2**, 6.2.2.4.2, 8.2.2

BUSY_STATUS 5.2.3, **6.2.1.7.1**, 6.2.1.8

CACHE **8.2.1**, 8.2.3

CACHE_AGENT **8.2.1**, 8.2.2, 8.2.3, 8.2.5

CACHE_CAPABLE **7.2.4.1.1**

CACHE_CMD **6.2.1.4**, 6.2.2.4.5

CACHE_OP_CMPLT **8.2.2**

CACHED **8.2.2**, 8.2.3–8.2.5

CANCEL **5.2.4**, 5.2.7.2.2, 5.2.8.1, 5.2.8.4, 5.2.8.5, 5.2.9.1, 5.2.9.4

CAPABILITY_ERROR **6.2.1.7.2**

CENTRAL_ARBITER 4.2.2.1, 4.2.2.2, 5.2.1–5.2.3, 5.2.7.1.1, 5.2.7.1.3, **7.2.4.2.1**

CENTRAL_MESSAGE_REQUEST **5.2.1**, 5.2.4, 5.2.7.1.1, 5.2.9.1, 5.2.9.2, 5.2.9.6, 7.2.4.2.1

CENTRAL_MESSAGE_SENT **5.2.1**, 5.2.9.1, 5.2.9.2, 5.2.9.6

CENTRALIZED_MASTER **4.2.1**, 4.2.2.3, 6.2.1.1

COMPARE_SWAP **6.2.1.5**, 6.2.1.8, 6.2.3.3–6.2.3.8, 7.2.4.1.1

COMPARE_SWAP_CAPABLE **7.2.4.1.1**

COMPELLED **6.2.1.1**, 6.2.2.3.2

COMPELLED_STATUS **6.2.1.1**, 6.2.1.6, 6.2.1.7.2, 6.2.1.8, 6.2.2.2.1, 6.2.3.1, 6.2.3.2, 6.2.3.22, 6.2.3.23

COMPETE **5.2.5**, 5.2.7.3.1–5.2.7.3.9, 5.2.8.2, 5.2.8.5, 5.2.9.2, 5.2.9.5

COMPETITOR **5.2.4**, 5.2.5, 5.2.6, 5.2.7.3.1–5.2.7.3.9, 5.2.8.2–5.2.8.6, 5.2.9.2–5.2.9.6

CONNECTION_COMMAND_PARITY_ERROR **6.2.1.7.2**, 6.2.1.8

CONNECTION_INFO_CAPTURED 6.2.1.7.1, **6.2.1.8**, 6.2.2.1.1

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

CONNECTION_PHASE 6.2.1.1–6.2.1.4, 6.2.1.7.2, **6.2.1.8**, 6.2.2.1.1, 6.2.2.1.6, 6.2.2.2.1–6.2.2.2.8, 6.2.2.3.1, 6.2.2.5.1, 6.2.2.5.3

CONNECTION_STATUS 6.2.1.1, 6.2.1.2, **6.2.1.8**, 6.2.2.4.1, 6.2.3.2

COPY_BACK **6.2.1.4**, 6.2.1.8, 6.2.2.5.1, 6.2.3.1, 8.2.2–8.2.4

DATA_COMMAND_PARITY_ERROR **6.2.1.7.2**

DATA_INFO_CAPTURED 6.2.1.1, 6.2.1.4, 6.2.1.7.1, **6.2.1.8**, 6.2.2.1.1, 6.2.2.1.4–6.2.2.1.6, 6.2.3.3, 6.2.3.5, 6.2.3.7, 6.2.3.9, 6.2.3.11, 6.2.3.13, 6.2.3.15, 6.2.3.17

DATA_LENGTH_0 **6.2.1.6**, 6.2.2.2.6

DATA_LENGTH_1 **6.2.1.6**, 6.2.2.2.7

DATA_LENGTH_2 **6.2.1.6**, 6.2.2.2.8

DATA_PARITY_ERROR **6.2.1.7.2**

DATA_PHASE 6.2.1.1, 6.2.1.4–6.2.1.6, 6.2.1.7.2, **6.2.1.8**, 6.2.2.1.4, 6.2.2.2.1–6.2.2.2.3, 6.2.2.2.5–6.2.2.2.8, 6.2.2.5.1–6.2.2.5.3, 6.2.3.2, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.16, 6.2.3.18

DATA_TAG_PARITY_ERROR **6.2.1.7.2**

DATA_WIDTH_0 **6.2.1.3**, 6.2.1.8, 6.2.2.2.6, 6.2.2.5.1, 6.2.2.5.2, 6.2.2.5.3

DATA_WIDTH_1 **6.2.1.3**, 6.2.1.8, 6.2.2.2.7, 6.2.2.5.1, 6.2.2.5.2, 6.2.2.5.3

DEPOSITION **5.2.3**, 5.2.4, 5.2.8.1, 5.2.8.4

DISC_INFO_CAPTURED **6.2.1.8**, 6.2.2.1.2, 8.2.2

DISCONNECTION_COMMAND_PARITY_ERROR **6.2.1.7.2**

DISCONNECTION_DATA_PARITY_ERROR **6.2.1.7.2**

DISCONNECTION_END 6.2.1.7.2, **6.2.1.8**, 6.2.2.3.1–6.2.2.3.3, 6.2.2.4.2–6.2.2.4.8, 6.2.3.23, 7.2.1

DISCONNECTION_PHASE 6.2.1.4–6.2.1.6, **6.2.1.8**, 6.2.2.2.1, 6.2.2.2.2, 6.2.2.2.5–6.2.2.2.8, 6.2.2.5.1, 6.2.2.5.3, 6.2.3.2, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.16, 6.2.3.18, 6.2.3.22

DISCONNECTION_TAG_PARITY_ERROR **6.2.1.7.2**

DISTRIBUTED_ARBITER_REQUEST 5.2.2, **5.2.3**, 5.2.4, 5.2.7.1.1, 5.2.8.1, 5.2.8.2, 5.2.8.6, 7.2.4.2.2

DISTRIBUTED_GRANT **5.2.3**

DISTRIBUTED_MASTER **5.2.3**, 5.2.8.1, 5.2.8.6, 6.2.1.1

DISTRIBUTED_MESSAGE **5.2.2**, 5.2.3, 5.2.4, 5.2.8.1, 5.2.8.4

DISTRIBUTED_MESSAGE_ENABLE 5.2.1, 5.2.2, **7.2.4.2.1**

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

DISTRIBUTED_MESSAGE_REQUEST **5.2.2**, 5.2.3, 5.2.4, 5.2.7.1.1, 5.2.8.1, 5.2.8.2, 5.2.8.6, 7.2.4.2.1

DISTRIBUTED_MESSAGE_SENT **5.2.2**, 5.2.8.1, 5.2.8.2, 5.2.8.6

END_OF_DATA 6.2.1.7.1, **6.2.1.7.2**, 6.2.2.4.2

END_OF_DATA_STATUS **6.2.1.7.1**, 6.2.1.8

ENTRANT **7.2.1**, 7.2.2

EXCLUSIVE_MODIFIED 8.2.2, **8.2.3**

EXCLUSIVE_MODIFIED_STATUS **8.2.2**, 8.2.5

EXCLUSIVE_UNMODIFIED 8.2.2, **8.2.3**

EXCLUSIVE_UNMODIFIED_STATUS **8.2.2**

FETCH_ADD_CAPABLE **7.2.4.1.1**

FETCH_ADD_BIG **6.2.1.5**, 6.2.1.8, 6.2.3.3–6.2.3.6, 7.2.4.1.1

FETCH_ADD_LITTLE **6.2.1.5**, 6.2.1.8, 6.2.3.3–6.2.3.6, 7.2.4.1.1

HIGH_SPEED_ENABLE 6.2.1.1, **7.2.4.2.2**

ILLEGAL_OPERATION_ERROR **6.2.1.7.2**

INIT 4.2.1, 4.2.2.1, 4.2.2.2, 5.2.1–5.2.5, 5.2.7.1.1–5.2.7.1.3, 6.2.1.1–6.2.1.6, 6.2.1.7.1, 6.2.1.7.2, 6.2.1.8, 6.2.2.1.1–6.2.2.1.6, 6.2.2.2.1–6.2.2.2.8, 6.2.2.3.1–6.2.2.3.3, 6.2.2.4.1–6.2.2.4.8, 6.2.2.5.1–6.2.2.5.3, 6.2.2.5.6, **7.2.1**, 9.2.1

INTERVENTION 6.2.1.7.2, 6.2.2.4.6, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.14, 6.2.3.16, 6.2.3.18, **8.2.2**

INTERVENTION_STATUS **6.2.1.7.1**, 6.2.2.2.5

INVALID 8.2.2, **8.2.3**

INVALID_STATUS **8.2.2**, 8.2.3, 8.2.5

INVALIDATE **6.2.1.4**, 6.2.1.8, 6.2.2.5.1, 8.2.2–8.2.5

INVALIDATE_COLLISION **8.2.5**

KEEP_COPY **8.2.2**, 8.2.3

LAST_TRANSACTION_FLAG **6.2.1.4**, 8.2.2

LIVE_INSERTION **7.2.1**

LIVE_WITHDRAWAL_COMMAND **7.2.1**, 7.2.3.5

LOCK_CMD_0 **6.2.1.5**, 6.2.1,6, 6.2.2.2.1

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

LOCK_CMD_1 **6.2.1.5**, 6.2.1,6, 6.2.2.2.2

LOCK_CMD_2 **6.2.1.5**, 6.2.1,6, 6.2.2.2.3

LOCK_HOLD 6.2.1.2, **6.2.1.5**, 6.2.2.2.1, 6.2.3.23

LOCKED 6.2.1.1, **6.2.1.5**, 6.2.3.2

LOW_SPEED **6.2.1.1**, 6.2.2.3.1

LOW_SPEED_STATUS **6.2.1.1**

MAILBOX_INITIALIZATION **9.2.1**

MAILBOX_SELECTED 6.2.1.7.2, 6.2.2.4.5, **9.2.1**

MASK_SWAP_CAPABLE **7.2.4.1.1**

MASK_SWAP **6.2.1.5**, 6.2.3.3–6.2.3.8, 7.2.4.1.1

MASTER 5.2.3, **6.2.1.1**, 6.2.1.2–6.2.1.6, 6.2.1.7.1, 6.2.1.7.2, 6.2.1.8, 6.2.2.1.1, 6.2.2.1.4, 6.2.2.2.1–6.2.2.2.9, 6.2.2.5.1–6.2.2.5.3, 6.2.2.5.6, 6.2.3.1, 6.2.3.3, 6.2.3.5, 6.2.3.7, 6.2.3.9, 6.2.3.11, 6.2.3.13, 6.2.3.15, 6.2.3.17, 6.2.3.19, 6.2.3.22, 6.2.3.24, 7.2.4.1.1, 8.2.2–8.2.5, 9.2.1

MASTER_CAPABILITY_ERROR **6.2.1.7.1**, 6.2.1.8

MASTER_CAPABLE **7.2.4.1.1**

MASTER_ENABLE 4.2.2.1, 4.2.2.2, 5.2.3, **7.2.4.2.2**

MEMORY **8.2.1**

MEMORY_AGENT **8.2.1**, 8.2.2, 8.2.4, 8.2.5

MESSAGE_BUSY 6.2.1.7.2, **9.2.1**

MESSAGE_CAPABLE **7.2.4.1.1**

MESSAGE_CONFLICT 6.2.2.4.5, **9.2.1**

MODIFIED_RESPONSE **6.2.1.4**, 6.2.1.7.2, 6.2.2.5.1, 6.2.3.1, 8.2.2–8.2.5

MORE **6.2.1.4**, 6.2.2.2.2

MULTIPLE_PACKET_CAPABLE **7.2.4.1.1**

MULTIPLE_PACKET_MODE **6.2.1.1**, 6.2.1.4, 6.2.1.7.1, 6.2.1.7.2, 6.2.1.8, 6.2.3.1, 6.2.3.2, 8.2.2

MULTIPLE_PACKET_MODE_ENABLE 6.2.1.1, **7.2.4.2.1**

NO_ARBITRATION_OWNER **5.2.3**, 5.2.7.1.3, 5.2.8.5, 5.2.8.6

PACKET_CAPABLE 6.2.1.1, **7.2.4.1.1**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

PACKET_ENABLED **6.2.1.1**

PACKET_LENGTH_2_CAPABLE **7.2.4.1.1**

PACKET_LENGTH_2_ENABLE 6.2.1.1, **7.2.4.2.1**

PACKET_LENGTH_4_CAPABLE **7.2.4.1.1**

PACKET_LENGTH_4_ENABLE 6.2.1.1, **7.2.4.2.1**

PACKET_LENGTH_8_CAPABLE **7.2.4.1.1**

PACKET_LENGTH_8_ENABLE 6.2.1.1, **7.2.4.2.1**

PACKET_LENGTH_16_CAPABLE **7.2.4.1.1**

PACKET_LENGTH_16_ENABLE 6.2.1.1, **7.2.4.2.1**

PACKET_LENGTH_32_CAPABLE **7.2.4.1.1**

PACKET_LENGTH_32_ENABLE 6.2.1.1, **7.2.4.2.1**

PACKET_LENGTH_64_CAPABLE **7.2.4.1.1**

PACKET_LENGTH_64_ENABLE 6.2.1.1, **7.2.4.2.1**

PACKET_NAK **6.2.1.7.2**, 6.2.2.4.2

PACKET_REQUEST **6.2.1.1**, 6.2.1.7.2, 6.2.2.2.1

PARITY_REPORT_ENABLE 5.2.6, 6.2.1.7.2, **7.2.4.2.2**

PARTIAL **6.2.1.4**, 6.2.1.7.2, 6.2.1.8, 6.2.2.5.1–6.2.2.5.3, 6.2.3.3–6.2.3.8

PARTICIPATING 6.2.1.1–6.2.1.6, **6.2.1.7.2**, 6.2.1.8, 6.2.2.1.5, 6.2.2.1.6, 6.2.2.5.1–6.2.2.5.3, 6.2.3.2, 8.2.2

PHASE_0 5.2.1–5.2.4, **5.2.5**, 5.2.6, 5.2.7.1.1, 5.2.7.2.1, 5.2.7.2.2

PHASE_1 5.2.1–5.2.4, **5.2.5**, 5.2.7.1.1, 5.2.7.1.3

PHASE_2 **5.2.5**, 5.2.6, 5.2.7.1.2, 5.2.8.3, 5.2.9.3

PHASE_3 5.2.2–5.2.4, **5.2.5**, 5.2.6, 5.2.7.1.1, 5.2.7.1.2, 5.2.7.2.1

PHASE_4 5.2.4, **5.2.5**, 5.2.6, 5.2.7.1.3, 5.2.8.5, 5.2.9.5

PHASE_5 5.2.1–5.2.4, **5.2.5**, 5.2.7.1.2, 5.2.7.1.3, 6.2.1.5, 7.2.1

POWER_UP 4.2.2.4, 5.2.7.1.3, 6.2.2.1.3, **7.2.1**, 7.2.3.1, 7.2.4.2.1–7.2.4.2.6

POWER_UP_TIME **7.2.1**, 7.2.3.2

PREVENT_EU **8.2.2**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

QUEUED_TRANSACTION_FLAG **8.2.2**, 8.2.3

READ_INVALID **6.2.1.4**, 6.2.1.8, 6.2.3.1, 8.2.2–8.2.5

READ_LOCKED **6.2.1.4**, 6.2.1.5, 6.2.1.8, 6.2.3.1

READ_MODIFIED **6.2.1.4**, 6.2.1.8, 6.2.3.1, 8.2.2–8.2.5

READ_PARTIAL **6.2.1.4**, 6.2.1.8, 6.2.3.1

READ_PARTIAL_LOCKED **6.2.1.4**, 6.2.1.5, 6.2.1.5, 6.2.3.1

READ_RESPONSE 6.2.1.3, **6.2.1.4**, 6.2.2.5.1, 6.2.2.5.3, 6.2.3.1

READ_SHARED **6.2.1.4**, 6.2.1.8, 6.2.3.1, 8.2.2–8.2.5

READ_UNLOCKED **6.2.1.4**, 6.2.1.8, 6.2.2.2.2, 6.2.3.1

READY_FOR_WITHDRAWAL 6.2.3.2, **7.2.1**, 7.2.3.5

REQUEST_MAILBOX **9.2.1**

REQUEST_MAILBOX_BUSY **9.2.1**

REQUEST_MAILBOX_CONFLICT **9.2.1**

REQUEST_MAILBOX_SELECTED **9.2.1**

REQUESTER 6.2.1.7.2, 8.2.2, **8.2.4**

REQUESTER_EXCLUSIVE 8.2.2, 8.2.3, **8.2.4**

REQUESTER_EXCLUSIVE_STATUS **8.2.2**

REQUESTER_SHARED 8.2.2, 8.2.3, **8.2.4**

REQUESTER_SHARED_STATUS **8.2.2**

REQUESTER_STATUS **8.2.2**

REQUESTER_WAITING 8.2.2, **8.2.4**

REQUESTER_WAITING_STATUS **8.2.2**

REQUESTER_WRITE_INVALID 8.2.2, **8.2.4**

REQUESTER_WRITE_INVALID_STATUS **8.2.2**

RESET_COMPLETE **7.2.1**, 7.2.2

RESET_EVENT **7.2.1**

RESOURCE_LOCK **6.2.1.5**, 6.2.3.2, 6.2.3.23

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

RESPONDER 8.2.2, **8.2.5**

RESPONDER_EXCLUSIVE 8.2.2, **8.2.5**

RESPONDER_EXCLUSIVE_STATUS **8.2.2**

RESPONDER_INVALIDATE 8.2.2, **8.2.5**

RESPONDER_INVALIDATE_STATUS **8.2.2**, 8.2.5

RESPONDER_SHARED 8.2.2, **8.2.5**

RESPONDER_SHARED_STATUS **8.2.2**

RESPONDER_STATUS **8.2.2**

RESPONDER_WRITE_INVALID 8.2.2, **8.2.5**

RESPONDER_WRITE_INVALID_STATUS **8.2.2**

RESPONSE_MAILBOX **9.2.1**

RESPONSE_MAILBOX_BUSY **9.2.1**

RESPONSE_MAILBOX_CONFLICT **9.2.1**

RESPONSE_MAILBOX_SELECTED **9.2.1**

ROUND_ROBIN **5.2.3**, 5.2.4, 5.2.8.2, 5.2.8.6

RQ **5.2.1**, 5.2.9.2

SECOND_PASS 5.2.1–5.2.3, **5.2.4**, 5.2.8.1, 5.2.8.2, 5.2.8.4, 5.2.8.6, 5.2.9.2, 5.2.9.6

SECOND_PASS_REQUIRED 5.2.1–5.2.3, **5.2.4**, 5.2.8.1, 5.2.8.4, 5.2.8.6, 5.2.9.1, 5.2.9.4, 5.2.9.6

SELECTED 6.2.1.5, **6.2.1.7.2**, 6.2.2.4.3, 6.2.3.4, 6.2.3.6,.6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.14, 6.2.3.16, 6.2.3.18

SELECTED_STATUS **6.2.1.7.1**

SETTLED **5.2.5**, 5.2.8.1, 5.2.8.3, 5.2.9.1, 5.2.9.3

SHARED_RESPONSE **6.2.1.4**, 6.2.1.7.2, 6.2.2.5.1, 6.2.3.1, 8.2.2–8.2.5

SHARED_UNMODIFIED 8.2.2, **8.2.3**

SHARED_UNMODIFIED_STATUS **8.2.2**, 8.2.3

SLAVE_WRITE **6.2.1.4**, 6.2.1.8, 6.2.2.1.5, 6.2.2.1.6, 6.2.2.5.1–6.2.2.5.3, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.14, 6.2.3.16, 6.2.3.18

SNARF_DATA 6.2.2.4.4, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, **8.2.2**, 8.2.3

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

SPLIT **6.2.1.2**, 6.2.2.3.3, 8.2.3, 8.2.5

SPLIT_ACCEPTOR **7.2.4.1.1**, 7.2.4.2.1, 8.2.1

SPLIT_CACHED 6.2.1.2, **8.2.2**

SPLIT_ENABLE 6.2.1.2, **7.2.4.2.1**, 8.2.1

SPLIT_INITIATOR 6.2.1.2, **7.2.4.1.1**

SPLIT_STATUS **6.2.1.2**, 6.2.1.6, 6.2.1.8, 6.2.2.2.2, 6.2.2.5.1, 6.2.3.3, 6.2.3.22, 6.2.3.23, 8.2.3–8.2.5

SYS_RESET 4.2.2.4, **7.2.1**, 7.2.2, 7.2.3.1–7.2.3.4, 7.2.4.2.1–7.2.4.2.6, 8.2.3–8.2.5

TAG_CAPABLE **7.2.4.1.1**

TAG_ENABLE 6.2.2.5.5, **7.2.4.2.2**

TRANS_CMD_0 **6.2.1.4**, 6.2.2.2.1

TRANS_CMD_1 **6.2.1.4**, 6.2.2.2.2

TRANS_CMD_2 **6.2.1.4**, 6.2.2.2.3

TRANS_CMD_3 **6.2.1.4**, 6.2.2.2.4

TRANSACTION_BEGIN 5.2.4, 5.2.8.5, **6.2.1.8**, 6.2.2.1.1, 6.2.3.1

TRANSACTION_COMPLETE 5.2.7.1.3, 5.2.8.5, 6.2.1.1–6.2.1.6, 6.2.1.7.1, 6.2.1.7.2, **6.2.1.8**, 6.2.2.1.1, 6.2.2.1.3, 6.2.3.22, 6.2.3.23, 8.2.2, 9.2.1

TRANSACTION_ERROR **6.2.1.7.2**, 6.2.1.8, 6.2.2.4.1, 8.2.2

TRANSACTION_ERROR_STATUS **6.2.1.7.2**

TRANSACTION_FLAG_STATUS **6.2.1.7.1**, 6.2.2.2.2, 8.2.3

TRANSACTION_TIMEOUT **6.2.1.8**, 6.2.3.1, 6.2.3.3, 6.2.3.5, 6.2.3.7, 6.2.3.9, 6.2.3.11, 6.2.3.13, 6.2.3.15, 6.2.3.17, 6.2.3.22, 6.2.3.24, 7.2.4.2.5

TRANSFER_1 **6.2.1.6**, 6.2.1.8

TRANSFER_2 **6.2.1.6**, 6.2.1.8, 7.2.4.1.1

TRANSFER_4 **6.2.1.6**, 6.2.1.8, 7.2.4.1.1

TRANSFER_8 **6.2.1.6**, 6.2.1.8, 7.2.4.1.1

TRANSFER_16 **6.2.1.6**, 6.2.1.8, 7.2.4.1.1

TRANSFER_32 **6.2.1.6**, 6.2.1.8, 7.2.4.1.1

TRANSFER_64 **6.2.1.6**, 6.2.1.8, 7.2.4.1.1

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

TRANSFER_COUNT_ZERO 6.2.1.1, **6.2.1.8**, 6.2.2.1.1, 6.2.2.1.4, 6.2.3.1, 6.2.3.3, 6.2.3.5, 6.2.3.7, 6.2.3.9, 6.2.3.11, 6.2.3.15, 6.2.3.17

TRANSFER_UNRESTRICTED **6.2.1.6**

TWO_PASS **5.2.3**, 5.2.4, 5.2.8.2

UNIQUENESS_GREATER **5.2.3**, 5.2.8.1, 5.2.8.4

WAIT **6.2.1.7.2**, 6.2.2.4.8

WAIT_CACHED 6.2.1.7.2, **8.2.2**

WAIT_STATUS 5.2.3, **6.2.1.7.1**, 6.2.1.8, 8.2.3, 8.2.4

WINNER 5.2.1–5.2.4, **5.2.5,** 5.2.7.1.2, 5.2.7.1.3, 5.2.8.1, 5.2.8.3, 5.2.8.5, 5.2.8.6, 5.2.9.1, 5.2.9.3, 5.2.9.6

WRITE_CMD **6.2.1.4**, 6.2.1.6, 6.2.1.7.2, 6.2.2.2.5, 6.2.2.5.1, 6.2.2.5.2, 6.2.2.5.3, 6.2.3.3, 6.2.3.5, 6.2.3.7, 6.2.3.9, 6.2.3.11, 6.2.3.13, 6.2.3.15, 6.2.3.17, 6.2.3.22, 6.2.3.23

WRITE_INVALID **6.2.1.4**, 6.2.2.5.1, 6.2.3.1, 8.2.2–8.2.5

WRITE_LOCKED **6.2.1.4**, 6.2.1.5, 6.2.2.5.1, 6.2.3.1

WRITE_NO_ACKNOWLEDGE **6.2.1.4**, 6.2.2.5.1, 6.2.3.1, 9.2.1, 9.2.2.1–9.2.2.14

WRITE_PARTIAL **6.2.1.4**, 6.2.2.5.1, 6.2.3.1

WRITE_PARTIAL_LOCKED **6.2.1.4**, 6.2.1.5, 6.2.2.5.1, 6.2.3.1

WRITE_RESPONSE 6.2.1.3, **6.2.1.4**, 6.2.1.8, 6.2.2.5.1, 6.2.2.5.3

WRITE_STATUS **6.2.1.4**, 6.2.1.7.2, 6.2.3.4, 6.2.3.6, 6.2.3.8, 6.2.3.10, 6.2.3.12, 6.2.3.14, 6.2.3.16, 6.2.3.18

WRITE_UNLOCKED **6.2.1.4**, 6.2.2.2.2, 6.2.2.5.1, 6.2.3.1, 9.2.1, 9.2.2.1–9.2.2.14

2.8 Implementation mnemonics

Data sheets for products based on this International Standard shall use the following mnemonics to indicate capabilities:

A32	32-bit addressing implemented.
A64	64-bit addressing implemented.
CACHE	The cache coherence protocol is implemented.
CARB	A central arbiter interface is implemented.
DARB	A distributed arbiter interface is implemented.
D32	Data path width of 32 bits.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D64	Data path width of 64 bits.
D128	Data path width of 128 bits.
D256	Data path width of 256 bits.
PKTn	Able to use packet transfer mode at a clock rate of n MHz. If more than one frequency is supported, multiple PKTn mnemonics shall be listed.
MSG	Message passing protocol is implemented.
MSTR	Capable of becoming bus master.
MSLOCK	Able to service mask and swap locked commands.
CSLOCK	Able to service compare and swap locked commands.
FABLOCK	Able to service fetch and add big endian locked commands.
FALLOCK	Able to service fetch and add little endian locked commands.
SPLT	Able to service split transactions.
SPLTING	Able to service and initiate split transactions.
TAGn	Tag lines are implemented. The number of tag lines implemented is n.

A module able to become bus master, with a 64-bit data path width, 32-bit address capability, cache coherence mechanism, and able to service split transactions, would be A32/CACHE/D64/MSTR/SPLT.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3. Bus signaling environment

3.1 Description

This clause describes and specifies the signaling environment with which Futurebus+ implementations must comply. These specifications are *independent* of the signaling technology used: Futurebus+ may be implemented with any technology (TTL, Backplane Transceiver Logic, ECL, CMOS, or GaAs), provided the condition of incident wave switching and the specified constraints on skew are met. In order to ensure compatibility between products within a set described by a profile, particular logic families may be associated with specific profiles in IEEE Std 896.2-1991.

3.1.1 Incident wave switching

Incident wave switching is a requirement for all Futurebus+ systems. The requirement is as follows: The parameters of current, voltage, and characteristic impedance, under all conditions of module loading, shall always enable the first voltage step launched by a driver down the transmission line to exceed the threshold required to trigger correct reception by the receiver, on both positive and negative transitions, and shall guarantee that subsequent reflections do not re-cross the receiver thresholds. Typically, this is related to the ratio of the current capable of being driven by a device, to the capacitance it presents to a bus line when the driver is disabled.

3.1.2 Skew

Skew is the variance in the propagation time of two or more signals that may travel over different paths, but share a common source and destination. Two signal transitions can never be guaranteed to be exactly simultaneous. Therefore, the bus protocol must always allow for a skew between them.

A critical skew in a Futurebus+ system is measured between information signals and the synchronization signal that marks their validity on the bus. The sender module's designer must ensure that transmitted information signals arrive at the connection to the bus before the validating synchronization signal occurs. This skew requirement must take into account any skews introduced by the bus transmitters themselves, and the signaling medium.

In addition, the designer of the signaling medium must guarantee that the skew introduced by capacitive loading does not delay the information signals more than the synchronization signals. A module that is logically completely passive is still electrically active with regard to bus skew. The length of the PCB trace stubs, through-plated holes, and the transceivers themselves contribute to the loading.

The designer of receiving modules must also delay the synchronization signal to account for any skew in the bus receivers as the information is received. This will ensure that the cause-and-effect relationship between data is preserved as the signals are provided to the first synchronization barrier within the receiving module.

3.2 Specification

3.2.1 Skew

The bus shall exhibit a skew such that the propagation delays of the signal lines to the left of the relational operator in relationships 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, and 3.8 are greater than or equal to the propagation delays of the signal lines to the right of the relational operator in these relationships:

All modules shall restrict the capacitive loading skew such that the relative loading on the signal lines exhibited by the module meets the relationships 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, and 3.8 above.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(AS*, DS*) ≥ (AD[ ]*, D[ ]*, BP[ ]*, TG[ ]*, TP*, CM[ ]*, CP*) (3.1)

AK* ≥ ST[ ]* (3.2)

AI* ≥ CA[ ]*, ST[ ]* (3.3)

(DK*, DI*) ≥ (AD[ ]*, D[ ]*, BP[ ]*, TG[ ]*, TP*, ST[ ]*) (3.4)

(AP*, AQ*, AR*) ≥ (AB[ ]*, AC[ ]*, ABP*) (3.5)

DS* ≥ DK* (3.6)

AS* ≥ (AK* AI*) (3.7)

GR* ≥ PE* (3.8)

3.2.2 Glitch filters

Glitch filters shall be provided if necessary to filter out a fundamental transmission line effect called the wire-OR glitch. It is anticipated that they will be needed for **AK***, **AI***, **DK***, **DI***, **AP***, **AQ***, **AR***, and **RE***. This specification is written assuming they are needed. The document specifying the electrical environment will specify whether they are needed for a particular signaling environment. For example, IEEE Std 896.2-1991 mandates the use of glitch filters in systems implemented using the signaling environment of Bus Transceiver Logic.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4. Centralized arbitration

4.1 Description

When a module needs to send data to or obtain data from another module, it must first gain tenure of the parallel bus. Since two or more modules may seek to gain tenure of the bus at the same time, an arbitration process is used to restrict tenure of the bus to one module at a time.

The central arbiter scheme involves a starred request/grant signal set from each requesting module to a central arbiter facility, which resolves simultaneous requests and serializes grants to the requesting modules.

This clause describes a simple centralized arbitration protocol that modules may use to gain tenure of the bus. It allows high-speed operation and low request-to-grant latency. A distributed arbitration protocol is specified in clause 5.

4.1.1 Bus lines used for centralized arbitration

4.1.1.1 RQ0* request 0

A module asserts **rq0*** to request bus mastership at one of its two priorities. Modules maintain **rq0*** asserted until the last transaction of their bus tenure.

Modules may assert **rq0*** even if they are asserting **rq1***. This possibly allows an increase of their request priority. A module maintains both **rq0*** and **rq1*** asserted until it becomes bus master and is in its last transaction on the parallel bus, or until it receives **GR*** and the internal request has been cancelled. Modules release **rq0*** and **rq1*** simultaneously.

4.1.1.2 RQ1* request 1

A module asserts **rq1*** to request bus mastership at one of its two priorities. Modules maintain **rq1*** asserted until the last transaction of their bus tenure.

Modules may assert **rq1*** even if they are asserting **rq0***. This possibly allows an increase of their request priority. A module maintains both **rq0*** and **rq1*** asserted until it becomes bus master and is in its last transaction on the parallel bus, or until it receives **GR*** and the internal request has been cancelled. Modules release **rq0*** and **rq1*** simultaneously.

Modules should not perform low-priority transactions during a tenure that resulted from a high-priority request. Modules should release the bus and request it again at the lower priority to avoid priority inversion.

4.1.1.3 GR* grant

The central arbiter asserts **gr*** in response to the assertion of **RQ0*** or **RQ1*** to indicate to a module that it is the master elect. The module may then begin parallel bus transactions when it detects that **ET*** is released. The central arbiter maintains the assertion of **gr*** until both **RQ0*** and **RQ1*** from the master have been released.

4.1.1.4 PE* preempt

The central arbiter asserts **pe*** to indicate to the master that a preemptive condition exists. The central arbiter releases **pe*** when it releases **gr***. The central arbiter resolves any metastability problems on the central arbiter occurring if **PE*** is asserted near the time that the master releases its request.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Preemptive conditions would be defined by the central arbiter design. The central arbiter might assert **PE*** if other modules are requesting at equal or higher priority. A different central arbiter design could assert **PE*** if any other module was requesting the bus.

The central arbiter asserts **pe*** during power-up and system reset to indicate to monarch-capable modules that there is a central arbiter in the system. Modules that are not monarch-capable are commanded to use the central arbiter through the CENTRAL_ARBITER bit in the Logical Common Control register (7.1.2.2.1).

4.1.2 Centralized arbitration operation

The centralized arbitration competition shown in figure 4 proceeds as follows:

1) Modules A and B each assert their arbitration request. For this example, Module A's request is a higher priority than Module B's request.
2) The central arbiter asserts **gr*** to Module A.
3) The current master, Module A, asserts **et*** after beginning a transaction.
4) During its last transaction, Module A finishes its need for bus tenure and releases its request.
5) Upon detecting that Module A has released its request, the central arbiter releases the grant to Module A and asserts the grant to Module B. Module B, at that point, becomes the master elect.
6) The master, Module A, after releasing the parallel bus signals, releases **et*** to indicate to the master elect that it is the new master.
7) After beginning its transaction the current master, Module B, asserts **et***.
8) During its last transaction, Module B finishes its need for bus tenure and releases its request.
9) Upon detecting that Module B has released its request, the central arbiter releases the grant to Module B. Since no requests are pending, the central arbiter issues no grant.
10) When the current master, Module B, has released the parallel bus signals, it releases **et***.

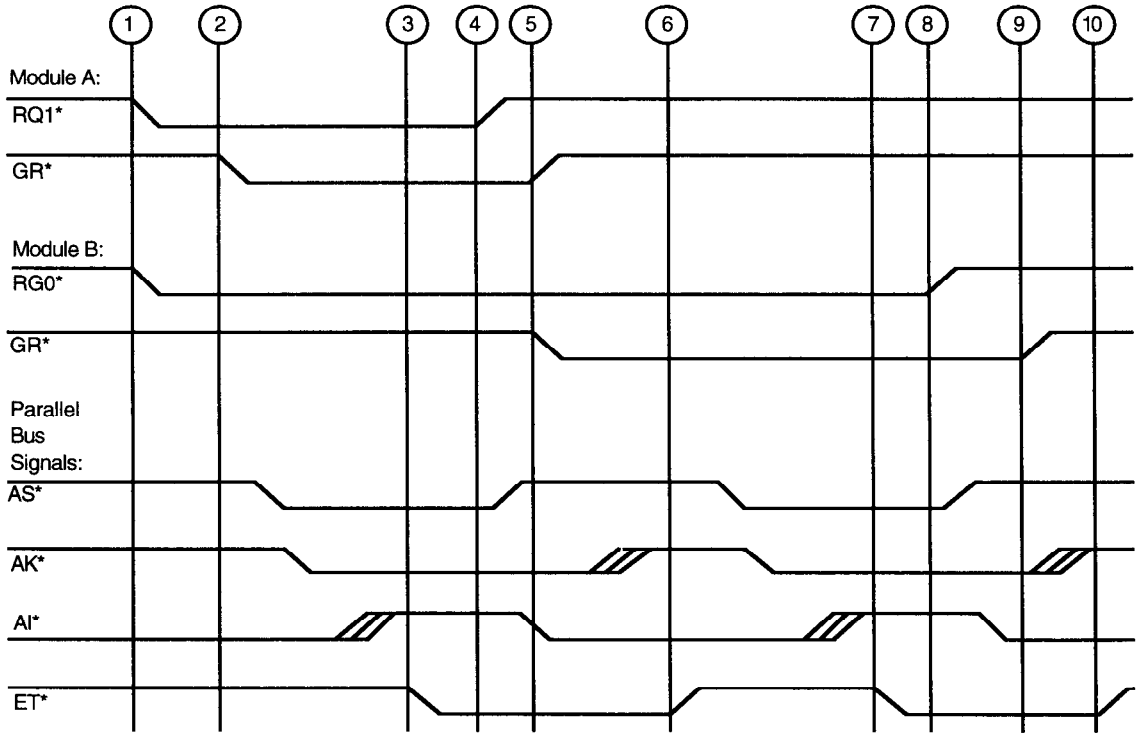


Figure 4—Two-competitor centralized arbitration

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.1.3 Central arbiter description

The central arbiter has few mandatory requirements. Since many different types of arbiters are possible that will allow interoperability, most requirements are the responsibility and privilege of the system integrator to specify.

Any number of bus allocation schemes may be implemented. Some possibilities may be:

— A simple, single-priority, first-come-first-served arbiter.
— A two-priority arbiter with fairness allocation within each priority level.
— A more complex scheme where each request is assigned one of 256 priorities. Priorities would be changed by modules sending messages to the central arbiter using the arbitrated message facilities.

4.2 Specification

4.2.1 Bus arbitration attributes

CENTRALIZED_MASTER. A module shall set CENTRALIZED_MASTER when ~INIT & (**rq0*** | **rq1***) & **GR*** & **~ET*** and maintain it set until INIT | **~rq0*** & **~rq1*** & **~et***.

4.2.2 Bus arbitration signals

4.2.2.1 RQ0*

A module may assert **rq0*** if ~INIT & MASTER_ENABLE & CENTRAL_ARBITER & **~GR*** to request mastership of the bus and shall maintain it asserted until INIT | **GR*** & **et*** during the master's last transaction of this tenure | **GR*** and the module chooses not to perform any transactions during this tenure.

4.2.2.2 RQ1*

A module may assert rq1* if ~INIT & MASTER_ENABLE & CENTRAL_ARBITER & **~GR*** to request mastership of the bus and shall maintain it asserted until INIT | **GR* & et*** during the master's last transaction of this tenure | **GR*** and the module chooses not to perform any transactions during this tenure.

4.2.2.3 GR*

The central arbiter asserts **gr*** to indicate to a module that it is the master elect. The central arbiter shall only assert **gr*** to one module at a time. The central arbiter shall release **gr*** to the CENTRALIZED_MASTER when it detects **~RQ1*** & **~RQ0***.

4.2.2.4 PE*

The central arbiter shall assert **pe*** if it detects a preemptive condition, and shall maintain it asserted until it releases **gr***. The central arbiter shall release **pe*** prior to asserting **gr*** to another module. If the central arbiter needs to assert **pe*** for consecutive tenures, it may choose to ignore the previous requirement and leave **pe*** asserted from the assertion of one **gr*** to the next.

The central arbiter shall assert **pe*** if POWER_UP | SYS_RESET and shall maintain it asserted for a minimum of 1 μs after it detects **REf** at logic zero.

NOTE—The detection of **PE*** asserted by the central arbiter is subject to metastability. The pulse width of **PE*** may be shorter than that required by a module. Modules must allow for these conditions and operate correctly.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5. Distributed arbitration and arbitrated messages

5.1 Description

When a module needs to send data to or obtain data from another module, it must first gain tenure of the parallel bus. Since two or more modules may seek to gain tenure of the bus at the same time, an arbitration process is used to restrict tenure of the bus to one module at a time. Futurebus+ provides the user with the choice of two methods to select the next bus master.

The central arbitration protocol is described in clause 4. The distributed arbitration protocol is described in this clause. In either case, the distributed protocol is used to broadcast arbitrated messages to one or more modules, or to the central arbiter.

5.1.1 Arbitrated messages—central arbiter

When the central arbiter is used, the arbitration process is used solely for the transmission of arbitrated messages. Two types of messages are provided, general arbitrated messages and central arbiter messages. General messages are usually directed at other modules on the bus, while central messages are used to control module request priorities in the central arbiter.

Some examples of uses for this messaging facility include sending power fail and other asynchronous broadcast interrupts, asynchronous targeted interrupts, software synchronization barriers, retry events, and bidding to supply services.

5.1.1.1 General arbitrated messages

Modules use a message number to resolve an arbitration competition. When two or more modules arbitrate to send a message, the winner is the one whose message number is greatest. Losing modules initiate a new arbitration competition to try again. The mechanism used is called parallel contention arbitration, a process whereby modules assert their message numbers on the arbitration bus and release signals according to an algorithm that, after a period of time, will ensure that only the winner's number remains on the arbitration bus.

General arbitrated messages are only 7 bits, so a unique number can be selected in a single competition. Each cycle consists of a competition using an 8-bit competition number, **cn[7..0].** A module wishing to deliver a general arbitrated message uses the competition number shown in table 2. The most significant bit is a 1, which ensures that general arbitrated messages have higher priority than central messages.

Table 2—General arbitrated message fields

General arbitrated messages (central arbiter)								
Bit	**cn7**	**cn6**	**cn5**	**cn4**	**cn3**	**cn2**	**cn1**	**cn0**
Pass 1	1	AM6	AM5	AM4	AM3	AM2	AM1	AM0

5.1.1.2 Central arbitrated messages

Central arbitrated messages are 14 bits, and are intended primarily to send priority information to the central arbiter. Central arbiter messages require two competition cycles in order to allow up to 256 priority levels to be used in a system. New message requests cannot be initiated between the first and second pass of a two-

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

pass message transmission. Although there can be more than one winner after the first pass, the second pass will choose a unique message.

A central arbiter message consists of three fields, as shown in table 3. The priority field, **PR[7..0]**, selects the priority level of the module's request line. This priority level is applied to the request line corresponding to the request select field, **RQ**. If **RQ** is zero, the **RQ0*** line is selected. If **RQ** is one, the **RQ1*** line is selected. The geographical address field, **GA[4..0],** identifies the module requesting the updated request level. In general, a module will control its own priority level, but it is possible for one module to take responsibility for another's priorities. The most significant bit of the competition number determines whether the message is in the first or last pass, and ensures that central arbitrated messages have lower priority than general ones.

Table 3—Central arbitrated message fields

Central arbitrated message (central arbiter)								
Bit	**cn7**	**cn6**	**cn5**	**cn4**	**cn3**	**cn2**	**cn1**	**cn0**
Pass 1	0	PR7	PR6	PR5	PR4	PR3	PR2	PR1
Pass 2	1	PR0	RQ	GA4	GA3	GA2	GA1	GA0

5.1.2 Arbitration and messages—distributed arbiter

Systems with no central arbiter use a distributed arbitration protocol. In this case, the arbitration process is used for both bus requests and arbitrated messages. The distributed arbitration requests allow modules to obtain bus tenure. The distributed arbitration messages are similar to the general arbitrated messages above, directed at other modules on the bus.

5.1.2.1 Distributed arbitration messages

Distributed arbitration messages are 7 bits consisting of bits **AM[6..0]**. However, they require a two-pass competition because a first pass of all 1's is needed to distinguish them from distributed arbitration requests, and to ensure that those requests do not interfere with the transmission of emergency information. A module wishing to deliver a distributed arbitration message uses the competition number shown in table 4. During the second pass, the most significant bit is set to 1 for consistency with the general arbitrated message format.

Table 4—Distributed arbitration message fields

Distributed arbitration messages (distributed arbiter)								
Bit	**cn7**	**cn6**	**cn5**	**cn4**	**cn3**	**cn2**	**cn1**	**cn0**
Pass 1	1	1	1	1	1	1	1	1
Pass 2	1	AM6	AM5	AM4	AM3	AM2	AM1	AM0

5.1.2.2 Distributed arbitration requests

The value of the distributed arbitration request numbers used by modules in a competition determine the number of competition passes. Some numbers require two passes of the control acquisition cycle, while others require only a single pass. Each pass of the control acquisition cycle consists of an arbitration competition using the 8-bit competition number, **cn[7..0]**. More than one module may have the highest number after

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

the first pass, but there is only a single winner at the end of the second pass. This is guaranteed because the competition numbers used in the second pass are unique. Arbitration numbers requiring only a single-control acquisition cycle may be freely and dynamically mixed with those requiring two-control acquisition cycles.

An arbitration number consists of three fields, as shown in table 5: the priority field, the round-robin field, and the geographical address field. A module's arbitration number maps onto one or two competition numbers. This mapping is purely a performance enhancement and does not imply that two separate classes of arbitration numbers exist. The bit **cn7** is used to indicate whether the module requires another arbitration pass, so that a single-pass arbitration number defeats a two-pass arbitration number if both occur in the same control acquisition cycle.

Table 5—Distributed arbitration request fields

Distributed arbitration requests (distributed arbiter)								
Bit	cn7	cn6	cn5	cn4	cn3	cn2	cn1	cn0
Single pass								
Pass 1	1	PRO	RR	GA4	GA3	GA2	GA1	GA0
Two pass								
Pass 1	0	PR7	PR6	PR5	PR4	PR3	PR2	PR1
Pass 2	1	PR0	RR	GA4	GA3	GA2	GA1	GA0

5.1.2.2.1 Priority field

The 8-bit priority field, **PR[7..0]**, selects one of 256 priority levels for the request. A module's priority class can be set by the system or the module. The highest two priority levels, where all but **PR0** are 1's, use the single pass competition cycle. This is possible because when **PR[7..1]** are all 1's the requester would always win a first pass, so it can be skipped for this case. Therefore, the priorities 254 and 255 have almost twice the request-to-grant performance of the other priorities, so they should be used in systems that only use one or two priority levels.

5.1.2.2.2 Round-robin field

The single-bit round-robin field, **RR**, in conjunction with the geographical address field, determines the position of a module in a round-robin request queue. The round-robin protocol ensures fair and equitable allocation of bus tenure to competing modules by allowing all modules a turn at bus tenure, in a sequence defined by the magnitude of the unique field assigned to that module.

Modules using the round-robin protocol monitor arbitration competitions. The value of a module's round-robin bit is adjusted each time a transfer of tenure to another module in its priority class is observed, whether or not it competes in that arbitration competition. The round-robin bit is set each time tenure of the bus is transferred to a module in the same priority class, but with a larger unique field value. Whenever tenure of the bus is transferred to a module in its priority class with a lesser unique field value, the round-robin bit is cleared.

Once a module has acquired tenure of the bus and cleared its round-robin bit, it is defeated in any subsequent arbitration competition by all modules competing with any geographical address field value including those that are lower. When all outstanding requests from modules with a lower value of geographical address field have been satisfied, it has a chance to regain bus tenure.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If a module using the round-robin protocol relinquishes its bus tenure to a module of higher priority class before its own transactions are complete, it may retain the value "1" for its round-robin bit, ensuring that it regains tenure of the bus as soon as it becomes available to a module of its priority class.

5.1.2.2.3 Geographical address field

The geographical address field is a five-bit field, **GA[4..0]**, which guarantees the uniqueness of the arbitration number. If a module is competing for the bus, this number must correspond to the geographical address of the module. The value "11111" is not allowed, due to a conflict with the CSR broadcast addressing scheme.

5.1.2.3 Deposing the master elect

In distributed arbitration, the module with the highest arbitration number at the end of the competition must wait until the master has finished performing bus transactions before it can become master. While waiting in this state, a module is referred to as the *master elect.*

There may be times when a module has urgent need of the bus after a master elect has been chosen, but before the master elect has become master. If that module has a higher arbitration number than the master elect, it may initiate a new competition to establish a new master elect. This process is referred to as *deposing the master elect.*

Deposing the master elect allows a high priority module to acquire tenure of the bus with minimum latency (although with some sacrifice to the overall performance of the system, since it forces a new arbitration competition).

5.1.2.4 Transaction preemption

Where a priority-based scheme is in operation, a module that is the initiator of a long block transaction or a series of related transactions that could take place within a single tenure of the bus may, in the presence of a master elect from a higher priority class, choose to voluntarily suspend its current transaction. This allows a scheme of preemptive bus scheduling to be implemented. The master obtains the information it needs on the new master elect by monitoring the identity and priority of arbitration winners while it simultaneously conducts its long transaction or series of transactions on the parallel bus.

If the master observes that a module of greater priority has become master elect, it should relinquish tenure of the bus to the higher priority module as soon as is practical.

5.1.2.5 Parking

If there has been no intervening bus request, a bus master retains control of the data bus even after it has finished its transactions. This state is known as parking. If the master has a new request while it still owns the data bus, it can begin a new data bus transaction immediately, without initiating an arbitration cycle. This has the potential to greatly reduce average bus latency in some types of systems.

5.1.3 Bus lines

5.1.3.1 AP*, AQ*, AR* control acquisition synchronization

These lines keep modules synchronized throughout the control acquisition process. The lines operate in a six-state sequence referred to as phase 0 through phase 5.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.1.3.2 AC[1..0]* arbitration condition

During phase 3 through phase 5, modules assert **AC0*** to signal an arbitration error.

In distributed arbitration, during phase 3 through phase 5, modules assert **AC1*** to inhibit the transfer of tenure at the completion of a control acquisition cycle. In a central arbiter system, modules assert **AC1*** to inhibit the transmission of a partial central arbitrated message which occurs during the first pass of the two-pass competition. **AC1*** is always asserted along with **AC0*** if an error occurs. If **AC1*** is asserted alone during phase 3 through phase 5, transfer of tenure is inhibited for reasons other than the detection of errors.

5.1.3.3 AB[7..0]* arbitration bus

Competing modules apply their competition numbers, **cn[ ]**, to **ab[ ]*** during phase 1. However, if any signal **cnx** is a logic zero, and the signal on the corresponding bus line **ABx*** is a logic one, then for as long as this condition persists, all bits, **ab[x-1..0]***, of lower significance are released. This continues through phase 3.

5.1.3.4 ABP* arbitration parity

If the number of ones in a competitors competition number is even, the competitor must assert **abp*** during phase 1. However, if any of the bits, **ab[ ]***, are released as described in 5.1.8.3, **abp*** shall also be released. This continues through phase 3.

5.1.4 Arbitration competition logic

As shown in the sample circuits in figures 5a) and 5b), if a module's competition number has a bit position of 0 and another module has a 1 in that position, the module with the 0 releases all **ab[ ]*** lines of lower significance. Eventually the **AB[ ]*** lines carry the number of the module with the highest arbitration number.

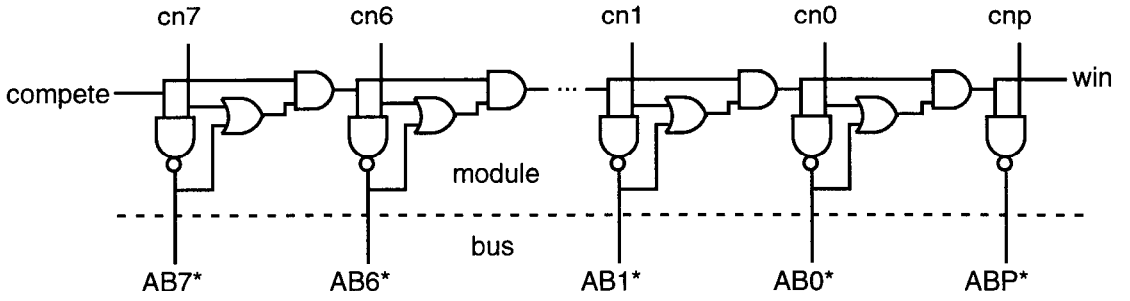
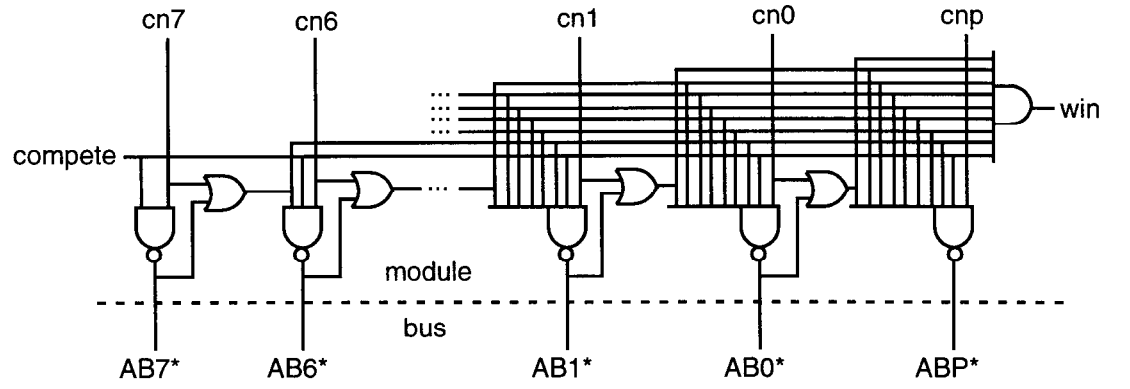


Figure 5a)—Serial implementation of parallel contention logic



NOTE—These circuits should not be considered a required implementation.

Figure 5b)—Parallel implementation of parallel contention logic

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

In some cases, modules release **ab[ ]*** lines only to assert them at a later time. Figure 6 illustrates an arbitration competition where three modules simultaneously begin competition using the numbers "00001010," "00001001," and "00000100," respectively. **AB[7..4]*** are not shown on the figure since they remain released throughout the sequence.

The arbitration competition in figure 6 proceeds as follows:

1) The three modules assert their competition numbers on the arbitration bus.
2) After a delay due to bus propagation, the bus lines become the wire-OR of the individual module lines causing **AB[3..0]*** to assert.
3) Module one, upon detecting **AB2*** asserted, releases **ab1***. Module two, upon detecting **AB2*** and **AB1*** asserted, releases **ab0***. Module three, upon detecting **AB3*** asserted, releases **ab2***.
4) The bus reflects the release of the bus lines.
5) Module 1, upon detecting **AB2*** released, asserts **ab1***. Module 2, upon detecting **AB2*** and **AB1*** released, asserts **ab0***.
6) The bus reflects the assertion of the bus lines.
7) Module 2 detects **AB1*** asserted and releases **AB0***.
8) The bus reflects the release of the bus line.

As shown in figure 6, the winning module (Module 1) releases **ab1*** and then later reasserts it. Module 1 also must wait for Module 2 to release **ab0*** before it knows it is the winner. The time required for the winning arbitration number to settle on the bus is referred to as the *arbitration competition settling time.*

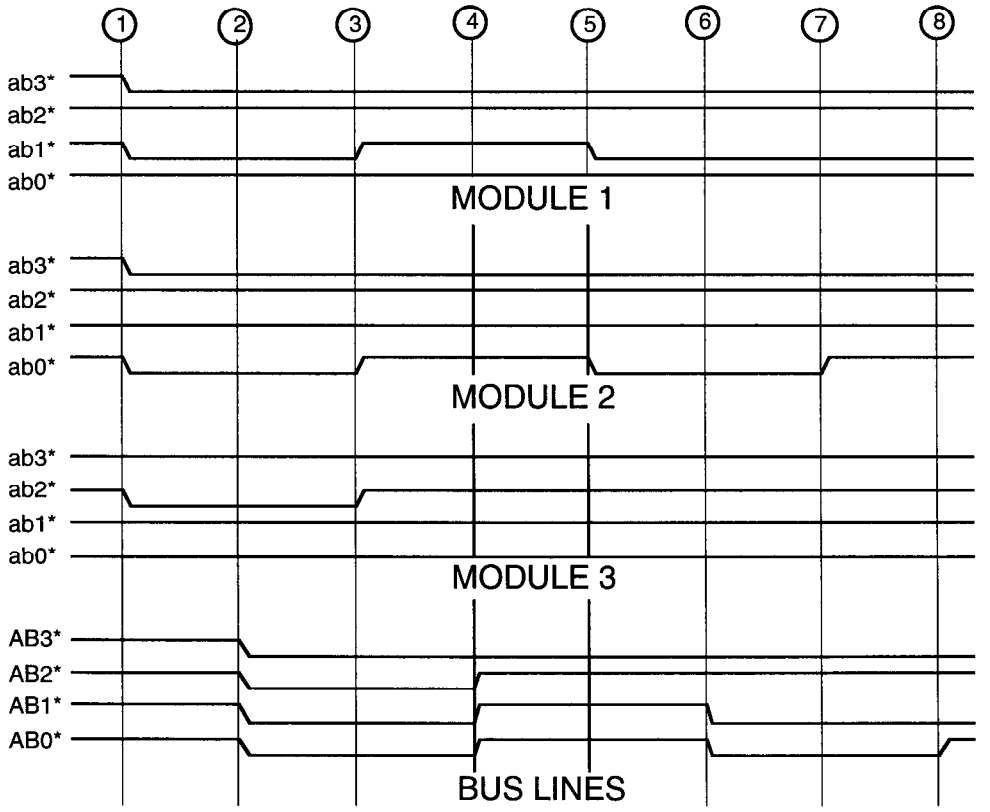


Figure 6—Arbitration competition example

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.1.5 Arbitration competition settling time

The arbitration settling time is a function of the length of the backplane, the module's competition number, and the worst-case transceiver and logic delays of both the internal arbitration logic of the module and that of its competitors. The theoretical calculation of the worst-case arbitration settling time can be extremely complex, and is beyond the scope of this International Standard. The worst possible settling time for any competition number, however, can be given by the following equation:

$$t_a = 10 * t_{pd'} + 5 * t_{ext} + 4 * t_{int} + t_{win}$$

The worst-case delay can be considerably lower for most competition numbers, so if the performance of the distributed arbiter is of concern, it is worth trying to optimize this equation for each module's requests and messages.

The symbol $t_{pd'}$ is used to represent the maximum end-to-end signal propagation delay along the transmission lines which constitute the logical bus. The value for $t_{pd'}$ is in the Bus Propagation Delay register described in 7.1, and is a function of the maximum length of the transmission line in the application system.

The symbol t_{int} is used to represent the delay between a change on any arbitration line, **ABx***, on the input to a module and the corresponding change on that module's adjacent arbitration output, **ab[x-1]***, or **abp***, as its competition number is applied to or withdrawn from those bits.

The symbol t_{ext} is used to represent the delay between a change on any arbitration line, **ABx***, on the input of any other competing module and the corresponding changes on that module's arbitration outputs, **ab[x-1]***, or **abp***, as its competition number is applied to or withdrawn from those bits.

The symbol t_{win} is used to represent the maximum delay between the time when the arbitration bus, **AB[7..0]***, becomes stable and equivalent to **cn[7.0]** on the input to a module and the indication within that module that it is the winner.

5.1.6 Arbitration states

The potential states that may be entered by a module involved in the arbitration process is illustrated in figure 7, along with the events that may result in a status change from each state. The figure shows examples of legal state transitions that may occur as a consequence of a control acquisition cycle.

Transitions from one state to another occur only when certain conditions are met.

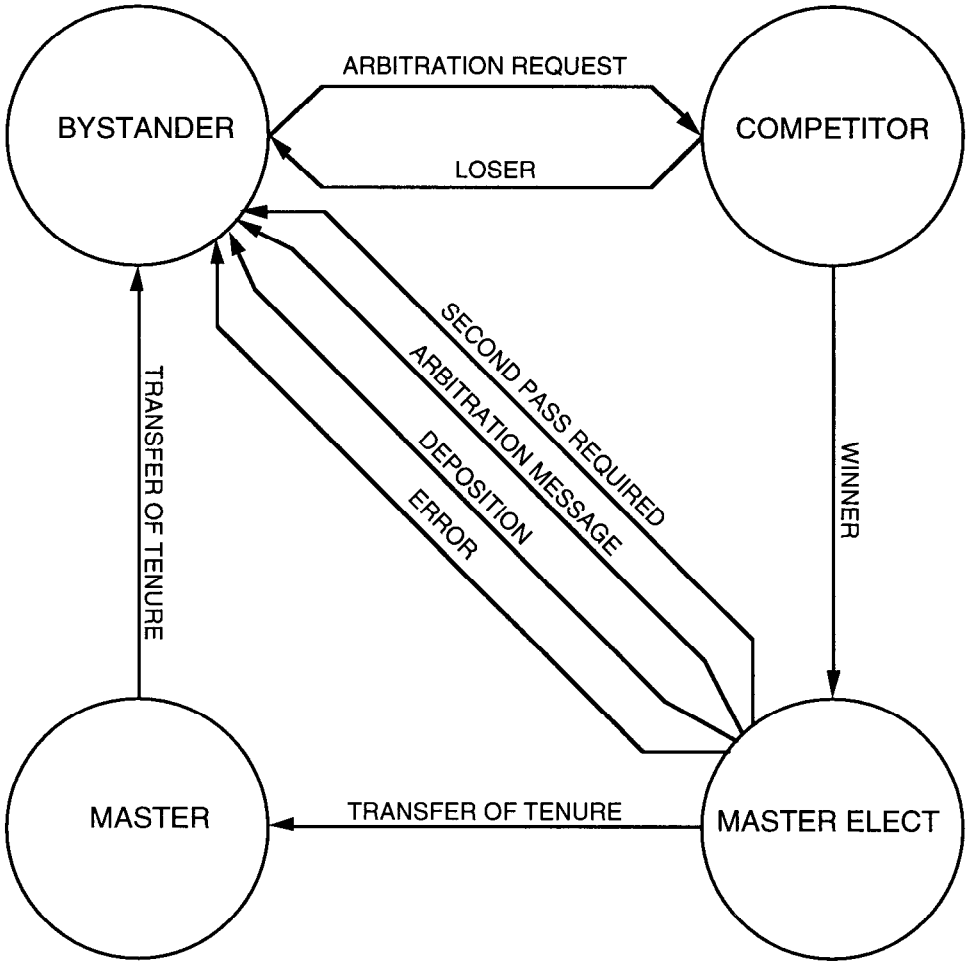
5.1.7 Arbitration phases

Six phases of the three synchronization signals, **AP***, **AQ***, and **AR***, are used during the control acquisition cycle. Normally, each phase is completed as quickly as propagation delays and metastability resolution times allow. If a two-pass arbitration competition is required, the entire control acquisition cycle is repeated.

The signals **AP***, **AQ***, and **AR*** are synchronizing signals representing the wire-OR of the module signals **ap***, **aq***, and **ar***, respectively. For any of them to be released on the bus, each module must individually release its signal. The release of a signal is a global synchronization point for each module. No module may proceed to the next phase of the synchronization arbitration cycle until all modules have released the appropriate arbitration line. Similarly, the assertion of one of these signals indicates that a new phase is starting, and all modules, even if they are not participating in the current arbitration competition, must remain synchronized by asserting their own signal in response.

Figure 8 shows the sequence of phases and line state changes.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale



NOTE—This figure is meant to provide a high-level overview, not a precise state diagram of the control acquisition process.

Figure 7—Control acquisition states

A control acquisition cycle consists of six phases that proceed sequentially, starting with phase 0 and ending with phase 5. The beginning and end of each phase is defined by a transition on one of the arbitration synchronizing signals, **AP***, **AQ***, and **AR***. These signals ensure that all modules, whether competing or not, remain synchronized throughout the control acquisition cycle.

5.1.7.1 Phase 0: idle phase

Phase 0, the idle phase, is identified by **AP*** released, **AQf** at logic zero, and **AR*** asserted. This is the state of the arbitration synchronization lines when no competition cycle is in progress. It may be the midpoint between two competition cycles during a two-pass request or message. Modules that wish to send a message or request the bus assert **ap***. This is the first indication to other modules that a competition is beginning.

5.1.7.2 Phase 1: decision phase

Phase 1, the decision phase, is identified by **AP*** asserted, **AQ*** released, and **AR*** asserted. Each module responds to the assertion of **AP*** with the assertion of **ap***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

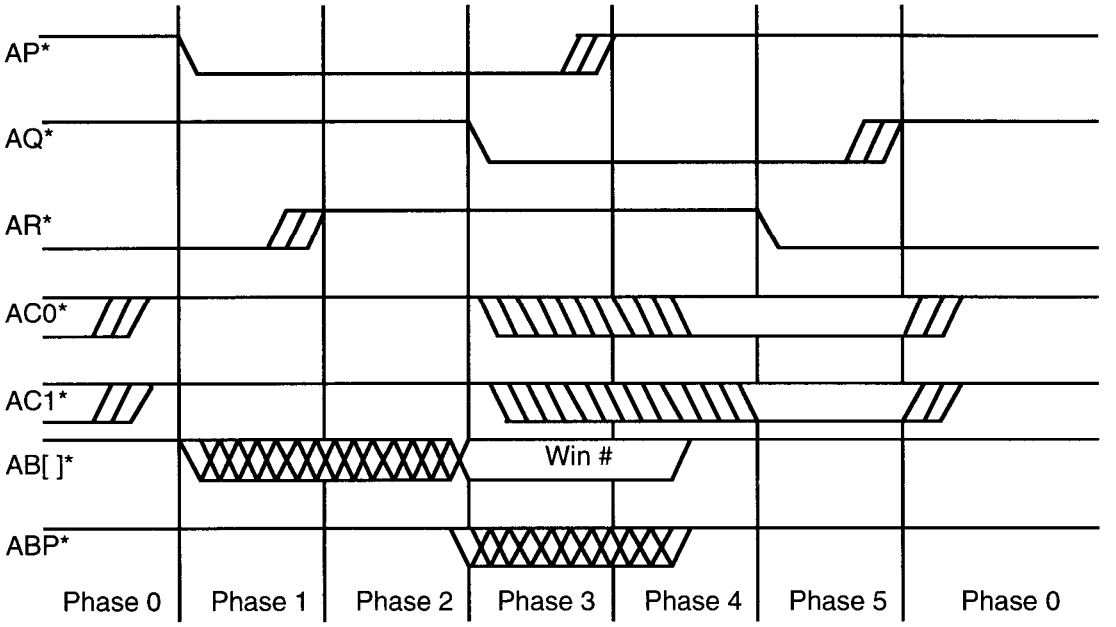


Figure 8—Control acquisition sequence

Since modules other than the initiator of the competition may decide to compete at the same time, the decision to compete is subject to metastability, due to the asynchronous relationship between the perceived assertion of **AP*** and a module's internal message or bus request signal to the arbitration logic. Modules must resolve any metastable condition before the decision phase is terminated.

After each module has resolved any metastability and made its decision whether or not to compete during this arbitration cycle, it releases **ar***. Noncompeting modules are synchronized with the arbitration cycle and may not initiate a new competition until the sequence has ended with return to phase 0.

No module may release **ar*** before it has released **ac0*** and **ac1***, and, if it is a competitor, it has enabled its competition number, **cn[ ]**, and competition number parity, **cnp**, onto the arbitration bus, **ab[ ]***, and **abp***.

5.1.7.3 Phase 2: competition phase

Phase 2, the competition phase, is identified by **AP*** asserted, **AQ*** released, and **ARf** at logic zero. Upon detecting the beginning of phase 2, competitors begin timing the arbitration competition settling time.

At the beginning of phase 2, all modules start a 1 μs to 2 μs arbitration error timer. This timer notifies the system of an arbitration error if it expires before the assertion of **AQ***. This provides a recovery mechanism for some types of failures that would otherwise cause the bus to lock up.

Each competitor monitors its internal "win" signal from the arbitration competition logic. Because the competition is an iterative process, the state of this competitor signal may change several times before settling to its final value. If a module's arbitration competition timer expires with its win signal asserted, it changes its state to winner (or master elect) and asserts **aq***.

If the 1 μs arbitration error timer expires before the assertion of **AQ***, there has been an arbitration error. All modules detecting the error assert **aq*** and ensure that the arbitration competition terminates without a transfer of tenure by asserting both **ac0*** and **ac1***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.1.7.4 Phase 3: error check

Phase 3, the error check phase, is identified by **AP*** asserted, **AQ*** asserted, and **AR*** released. All modules, upon observing **AQ*** asserted, assert **aq***.

During phase 3, all modules latch the value of the winning competition number from the bus and check its parity. Competition losers may also check that the winning competition number is greater than their own. Any modules detecting an error condition assert **ac0*** and **ac1***.

In a distributed arbiter system, if the winner of the arbitration is making a bus request, that module becomes the master elect. Phase 3 continues as long as the master's transaction on the parallel bus continues. Modules may request to depose the master elect by asserting **ac1*** during phase 3, causing the arbitration competition to terminate without a transfer of tenure. The completion of a two-pass arbitration competition takes precedence over an attempt to depose during the first pass. Note that the decision to depose the master elect is also subject to metastability.

In a distributed arbiter system, each module monitors **AS*** to detect its release by the master. This signals that the master is ending a bus transaction, and there may be a transfer of tenure. When **AS*** is released, each module releases **ap***.

NOTE—If **AS*** is asserted just as the bus enters phase 3, some modules may detect this condition, releasing **ap***, while others may not. Modules that have released **ap*** will be stuck, unable to depose the master elect, until the next release of **AS*** or assertion of **AC1***.

In a distributed arbiter system, modules sending arbitration messages signal that there is to be no transfer of tenure by asserting **ac1*** during phase 3.

In a central arbiter system, modules release **ap*** as soon as they have valid data on **ac0*** and **ac1***.

In either case, a module competing with a two-pass arbitration number and finding itself an intermediate winner asserts **ac1*** to prevent the premature transfer of message or tenure. Only when both passes have completed and there is a single winner is the message transmitted or tenure transferred. All modules also monitor **AC1***, and release **ap*** if they observe **AC1*** asserted. This indicates that the cycle has been canceled or is incomplete.

5.1.7.5 Phase 4: master release

Phase 4, the master release phase, is identified by **APf** at logic zero, **AQ*** asserted, and **AR*** released. At the beginning of phase 4 all modules release **ab[ ]***.

In a distributed arbiter system, all modules monitor **AC0*** and **AC1*** to determine when either an error is being reported, deposition is being requested, or tenure transfer is being inhibited.

In a distributed arbiter system, if the master wishes to perform additional transactions, it may cancel the transfer of tenure by asserting **ac1*** during phase 4. If it is ready to transfer tenure, and neither **AC0*** or **AC1*** are asserted, and **AKf** is at logic zero, it ensures that all information bus lines on the parallel bus are released. Then it asserts **ar***, signaling the transition to phase 5. If there is to be no transfer of tenure, the master may proceed immediately to phase 5 by asserting **ar***.

In a distributed arbiter system, each module, other than the master, starts a 1 μs timer when **APf** is detected at logic zero. Then it monitors **AR***, awaiting its assertion by the master. On observing the assertion of **AR***, a module should disable the timer and assert **ar***. If the 1 μs period expires due to some failure on the part of the master, the module should assert **ar*** to terminate phase 4.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

In a central arbiter system, all modules immediately assert **ar*** to exit phase 4.

5.1.7.6 Phase 5: tenure/message transfer

Phase 5, the message or tenure transfer phase, is identified by **AP*** released, **AQ*** asserted, and **AR*** asserted. All modules save the state of **AC0*** and **AC1*** during phase 5, to determine if a message or tenure transfer is to take place, or whether it has been prevented by either cancellation or the detection of an arbitration error. Once they have latched this status, they release **aq***.

In a distributed arbiter system, the master elect, upon detecting the assertion of **AR*** and observing both **AC0*** and **AC1*** released, assumes tenure of the bus. If either is asserted, the current master retains ownership of the bus.

If **AC1*** was asserted at the beginning of phase 5, message or tenure transfer is inhibited. This may be for any of the following reasons:

1) An arbitration error occurred.
2) A second pass competition is required before the acquisition cycle can be completed.
3) In a central arbiter system, a second pass competition is required before the central arbitrated message can be completed.
4) In a distributed arbiter system, an arbitration message has been sent.
5) In a distributed arbiter system, the master has decided to retain the bus for additional transactions.
6) In a distributed arbiter system, the new master elect is being deposed by a higher priority module.

Following the first pass of a two-pass competition, any module that did not compete in the first acquisition cycle must become inhibited from competing during the second pass of the control acquisition cycle. Any competitor that was an intermediate winner must compete in the second pass using a competition number appropriate to the second pass of a two-pass arbitration competition.

After the second pass of the control acquisition cycle, any module that was inhibited at the start of that cycle may remove that inhibition, allowing it to be a competitor in the following arbitration competition.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.1.8 Arbitration examples

5.1.8.1 Single-pass competition (central or distributed)

Figure 9 illustrates a single-pass arbitration competition which results in a transfer of tenure (in a distributed system) or general message transmission (in a central system). Note that the state of AS* is irrelevant in a central arbiter system. The operation proceeds as follows:

1) When a distributed bus request or general arbitrated message request is made by one or more modules, those modules assert **ap*** beginning phase 1.
2) Other modules, when they detect **AP*** asserted, decide whether or not to compete and assert **ap***.
3) Competing modules activate their competition number on **ab[ ]*** and **abp***, and release **ar***.
4) After competitors detect **ARf** at logic zero, indicating the beginning of phase 2, they begin timing the arbitration competition settling time.
5) After the arbitration lines have settled, one of the modules detects it is the winner and asserts **aq*** causing phase 3 to begin.
6) Modules then latch the identity of the winner from the **AB[ ]*** lines, check for errors, and in a distributed bus request, wait for the release of **AS***.
7) In a distributed bus request, when **AS*** is released all modules release **ap***. In a general message request, **ap*** is released immediately.
8) When modules detect **APf** at logic zero, phase 4 begins.
9) In a distributed bus request, when the master completes its final transaction, it asserts **ar*** and changes its state to bystander. In a general message request, **ar*** is asserted by all modules immediately.
10) In a distributed bus request, when the master elect detects **AR*** asserted, indicating the beginning of phase 5, it changes its state to master and may begin transactions on the parallel bus.
11) Modules then detect that a transfer of tenure or message transmission occurred and release **aq***.

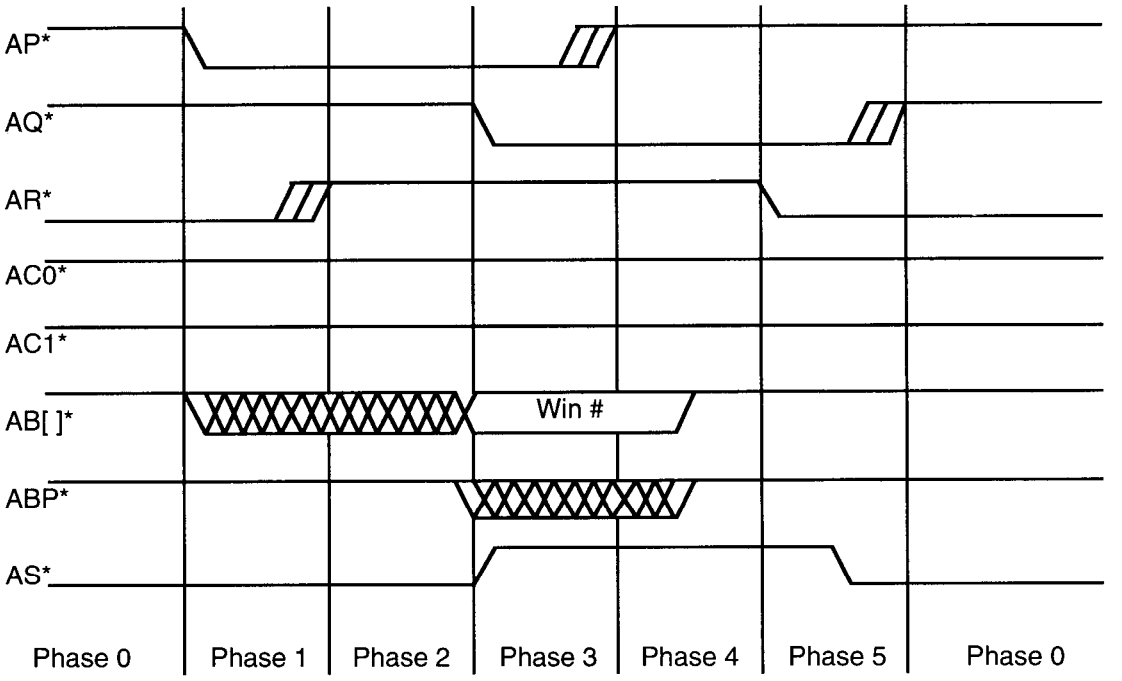


Figure 9—Single-pass competition

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.1.8.2 Single-pass deposed (distributed) competition

Figure 10 illustrates a single-pass distributed bus request competition which is deposed, cancelling the transfer of tenure. The operation proceeds as follows:

1) When an arbitration request is made by one or more modules those modules assert **ap*** beginning phase 1.
2) Other modules, when they detect **AP*** asserted, decide whether or not to compete and assert **ap***.
3) Competing modules activate their competition number on **ab[ ]*** and **abp***, and release **ar***.
4) After competitors detect **ARf** at logic zero, indicating the beginning of phase 2, they begin timing the arbitration competition settling time.
5) After the arbitration lines have settled one of the modules detects it is the winner and asserts **aq*** causing phase 3 to begin.
6) Modules then latch the identity of the master elect from the **AB[ ]*** lines, check for errors, and wait for the release of **AS***.
7) A noncompeting module then receives a bus request and determines that its competition number would have won the previous competition. It therefore asserts **ac1*** to depose the master elect and releases **ap***.
8) Other modules upon detecting **AC1*** asserted release **ap***.
9) When modules detect **AC1*** asserted and **APf** at logic zero, they assert **ar*** to begin phase 5.
10) Modules then detect a transfer of tenure did not occur and release **aq***.
11) When modules detect **AQf** at logic zero, they begin a new arbitration competition.

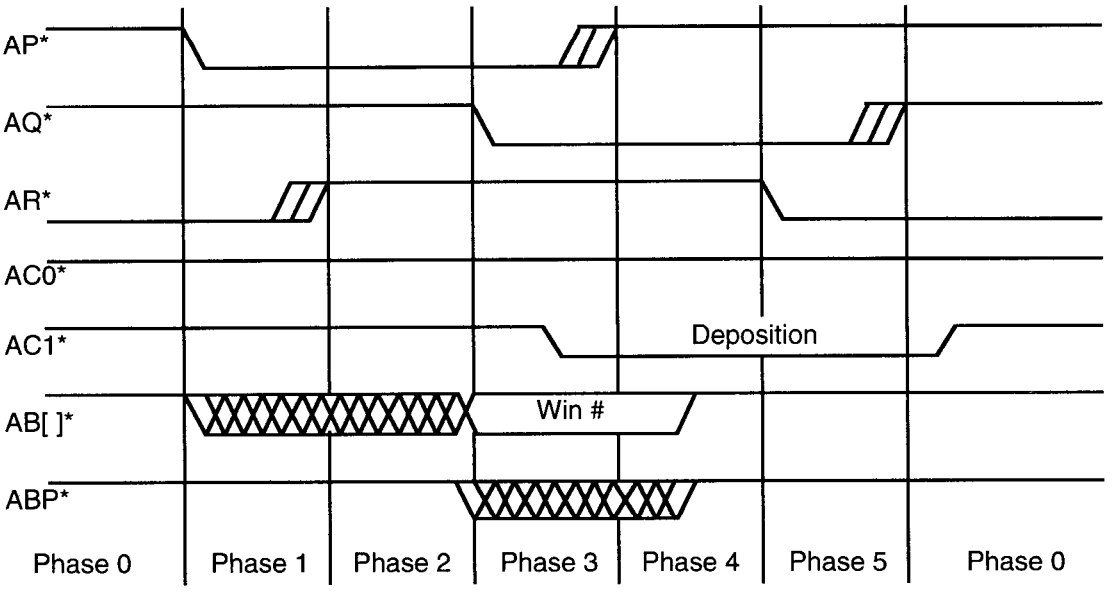


Figure 10—Single-pass deposed competition

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.1.8.3 Distributed arbitration message

Figure 11 illustrates a two-pass distributed arbitration competition resulting in an distributed arbitration message being sent. The operation proceeds as follows:

1) When an arbitration message request is made by one or more modules those modules assert **ap*** beginning phase 1.
2) Other modules, when they detect **AP*** asserted, decide whether or not to compete and assert **ap***.
3) Competing modules activate their competition number on **ab[ ]*** and **abp*,** and release **ar***.
4) After competitors detect **ARf** at logic zero, indicating the beginning of phase 2, they begin timing the arbitration competition settling time.
5) The modules sending arbitration messages quickly determine they are the winner and assert **aq*** causing phase 3 to begin.
6) The modules sending an arbitration message assert **ac1*** to cause a second pass competition to occur and release **ap***.
7) Other modules upon detecting **AC1*** asserted release **ap***.
8) When modules detect **AC1*** asserted and **APf** at logic zero they assert **ar*** to begin phase 5.
9) Modules then detect a transfer of tenure did not occur and that a second pass is required and release **aq***.
10) When modules that are not inhibited by losing the first pass of the competition detect **AQf** at logic zero, they begin a new arbitration competition.
11) The competition then proceeds with the module with the highest arbitration message number winning. The competition is again cancelled to prevent a transfer of tenure.

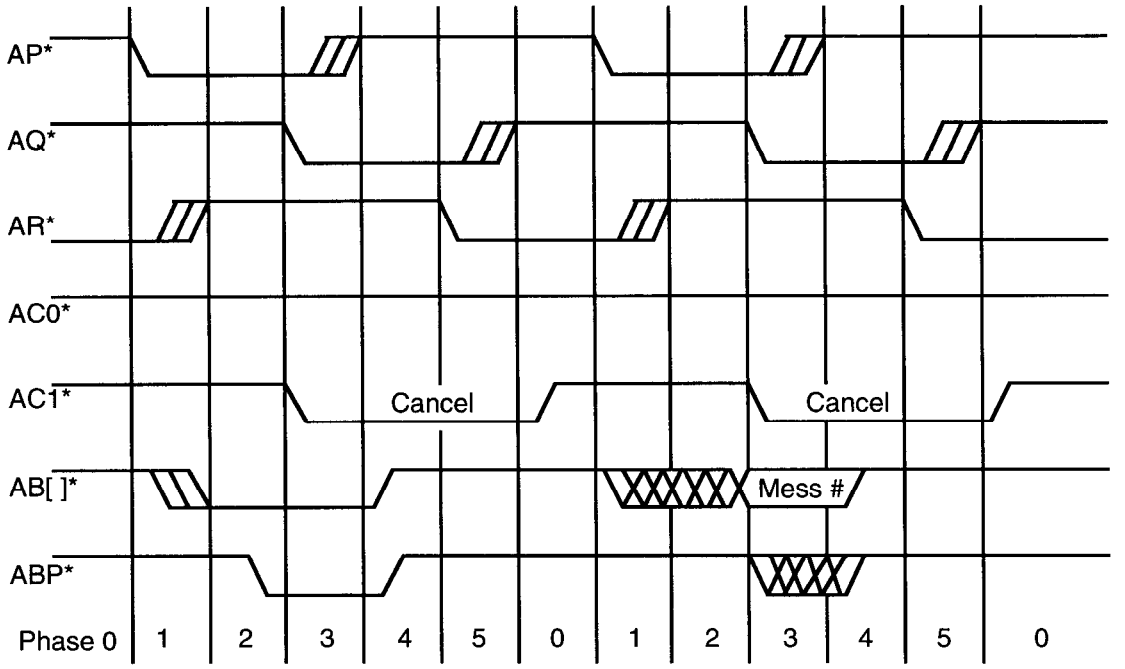


Figure 11—Arbitration message

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.1.8.4 Two-pass competition (central or distributed)

Figure 12 illustrates a two-pass arbitration competition that results in a transfer of tenure (in a distributed system) or a central message transmission (in a central system). The first pass consists of a single-pass competition which is canceled by the first-pass winners. Competitors losing the first pass are inhibited from competing in the second pass. The second pass operates identically to a single-pass competition with transfer of tenure as was described in 5.1.8.1 except that only first-pass winners compete. Note that the state of **AS*** is again irrelevant in a central arbiter system.

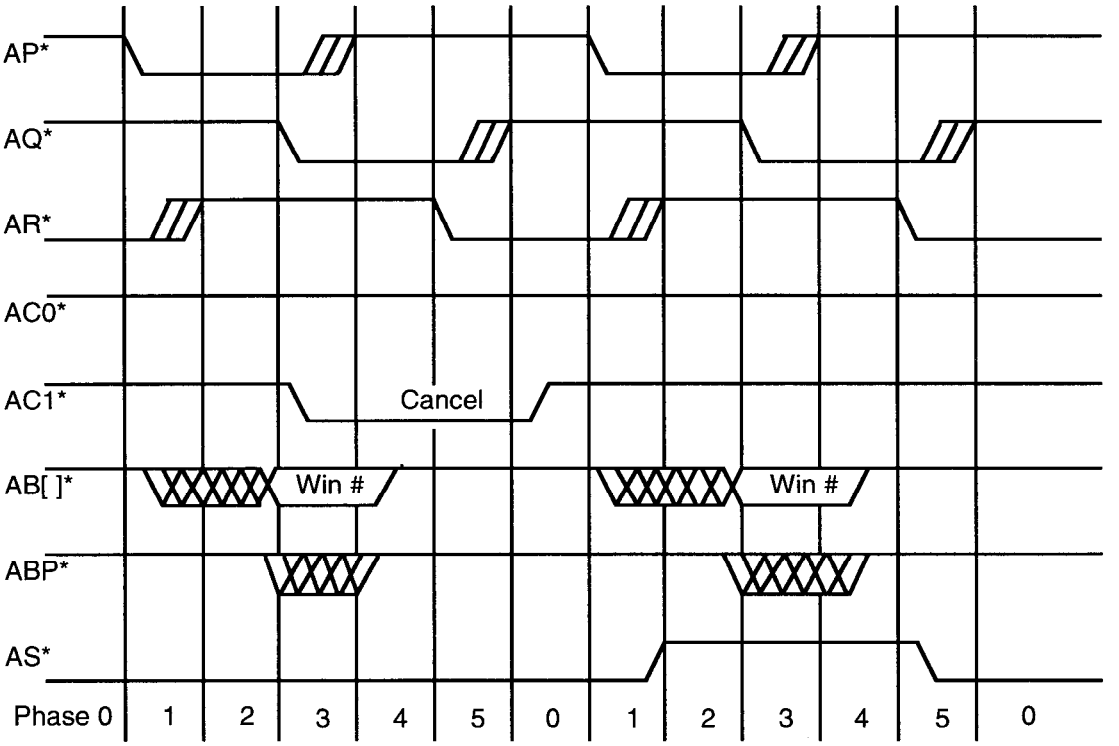


Figure 12—Two-pass competition

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.1.8.5 Single-pass competition with error

Figure 13 illustrates a single-pass arbitration competition in which an error is detected cancelling the transfer of tenure or transmission of the message. The operation proceeds normally until the error is detected in phase 3 causing one or more modules to assert both **ac0*** and **ac1***. The operation then proceeds in the same way as other canceled cycles except that modules take note of the error in phase 5 and take the appropriate error recovery action.

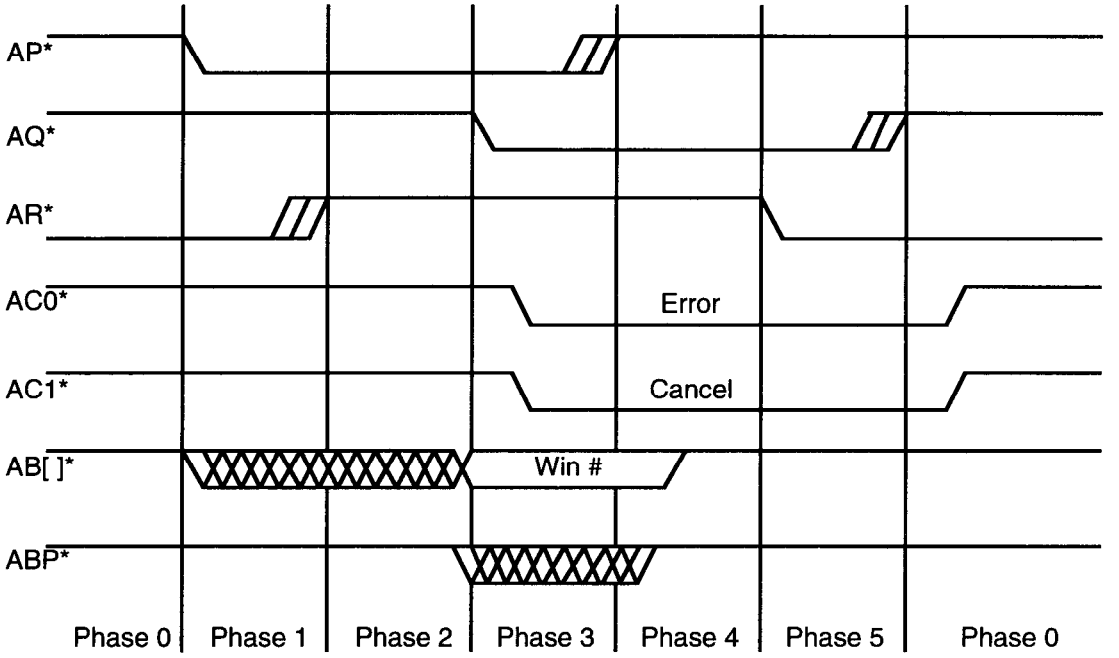


Figure 13—Single-pass competition with error

5.2 Specification

5.2.1 Arbitrated message attributes—central arbiter

ARBITRATED_MESSAGE_REQUEST. Modules may set ARBITRATED_MESSAGE_REQUEST when CENTRAL_ARBITER & ~INIT & DISTRIBUTED_MESSAGE_ENABLE & ~ARBITRATED_MESSAGE_SENT & ~CENTRAL_MESSAGE_REQUEST & ~SECOND_PASS & PHASE_0 & **~ap*** to send an arbitration message. Modules shall clear ARBITRATED_MESSAGE_REQUEST when INIT | (~CENTRAL_ARBITER | ARBITRATED_MESSAGE_SENT | ~DISTRIBUTED_MESSAGE_ENABLE) & ~SECOND_PASS & PHASE_0 & **~ap***.

NOTE—The decision to set ARBITRATED_MESSAGE_REQUEST is subject to metastability. Modules shall not release **ar*** until adequate time has been allowed for the resolution of this condition.

CENTRAL_MESSAGE_REQUEST. Modules may set CENTRAL_MESSAGE_REQUEST when CENTRAL_ARBITER & ~INIT & DISTRIBUTED_MESSAGE_ENABLE & ~CENTRAL_MESSAGE_SENT & ~ARBITRATED_MESSAGE_REQUEST & ~SECOND_PASS & PHASE_0 & **~ap*** to send an arbitration message. Modules shall clear CENTRAL_MESSAGE_REQUEST when ~CENTRAL_ARBITER | INIT | (CENTRAL_MESSAGE_SENT | ~DISTRIBUTED_MESSAGE_ENABLE) & ~SECOND_PASS & PHASE_0 & **~ap***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE—The decision to set CENTRAL_MESSAGE_REQUEST is subject to metastability. Modules shall not release **ar*** until adequate time has been allowed for the resolution of this condition.

ARBITRATED_MESSAGE_SENT. A module shall set ARBITRATED_MESSAGE_SENT when ARBITRATED_MESSAGE_REQUEST & WINNER & ~**AC0*** & PHASE_5. A module shall clear ARBITRATED_MESSAGE_SENT when ~ARBITRATED_MESSAGE_REQUEST.

CENTRAL_MESSAGE_SENT. A module shall set CENTRAL_MESSAGE_SENT when CENTRAL_MESSAGE_REQUEST & WINNER & ~SECOND_PASS_REQUIRED & ~**AC0*** & PHASE_5. A module shall clear CENTRAL_MESSAGE_SENT when ~CENTRAL_MESSAGE_REQUEST.

PR[7..0]. A module may load an 8-bit priority value into **PR[7..0]** to change central arbiter priority levels when ~CENTRAL_MESSAGE_REQUEST.

RQ. Modules may set **RQ** to change the priority level of central arbiter request line **RQ1***. Modules may clear **RQ** to change the priority level of central arbiter request line **RQ0***. Any change to **RQ** shall only occur when ~CENTRAL_MESSAGE_REQUEST.

AM[13..0]. A module shall load a 7-bit arbitrated message value into **AM[6..0]** when (PHASE_0 | PHASE_1) & ARBITRATED_MESSAGE_REQUEST & **ar***. A module shall load a 14-bit arbitrated message value into **AM[13..0]** when ~SECOND_PASS & (PHASE_0 | PHASE_1) & CENTRAL_MESSAGE_REQUEST & **ar***, where **AM[13..6]** shall equal **PR[7..0]**, **AM5** shall equal **RQ**, and **AM[4..0]** shall equal **GA[4..0].**

5.2.2 Arbitrated message attributes—distributed arbiter

DISTRIBUTED_MESSAGE_REQUEST. Modules may set DISTRIBUTED_MESSAGE_REQUEST when ~CENTRAL_ARBITER & ~INIT & DISTRIBUTED_MESSAGE_ENABLE & ~DISTRIBUTED_ARBITER_REQUEST & ~DISTRIBUTED_MESSAGE_SENT & ~SECOND_PASS & PHASE_0 & **~ap*** to send an arbitration message. Modules shall clear DISTRIBUTED_MESSAGE_REQUEST when CENTRAL_ARBITER | INIT | (DISTRIBUTED_MESSAGE_SENT |~DISTRIBUTED_MESSAGE_ENABLE) & ~SECOND_PASS & PHASE_0 & **~ap***.

NOTE—The decision to set DISTRIBUTED_MESSAGE_REQUEST is subject to metastability. Modules shall not release **ar*** until adequate time has been allowed for the resolution of this condition.

DISTRIBUTED_MESSAGE_SENT. A module shall set DISTRIBUTED_MESSAGE_SENT when DISTRIBUTED_MESSAGE_REQUEST & WINNER & ~SECOND_PASS_REQUIRED & ~**AC0*** & PHASE_5. A module shall clear DISTRIBUTED_MESSAGE_SENT when ~DISTRIBUTED_MESSAGE_REQUEST.

DISTRIBUTED_MESSAGE. A module shall set DISTRIBUTED_MESSAGE if ~CENTRAL_ARBITER & PHASE_3 & ~SECOND_PASS & **AB7*** & **AB6*** & **AB5*** & **AB4*** & **AB3*** & **AB2*** & **AB1*** & **AB0***. Modules shall maintain DISTRIBUTED_MESSAGE set until INIT | PHASE_0 & ~SECOND_PASS."

AM[6..0]. A module shall load an 7-bit arbitration message value into **AM[6..0]** when (phase_0 | phase_1) & DISTRIBUTED_MESSAGE_request & **ar***.

5.2.3 Arbitration attributes—distributed arbiter

DISTRIBUTED_ARBITER_REQUEST. Modules may set DISTRIBUTED_ARBITER_REQUEST when ~CENTRAL_ARBITER & ~INIT & MASTER_ENABLE & ~MASTER &

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

~DISTRIBUTED_MESSAGE_REQUEST & ~SECOND_PASS & PHASE_0 & **~ap*** to obtain bus mastership. Modules shall clear DISTRIBUTED_ARBITER_REQUEST when CENTRAL_ARBITER | INIT | (MASTER_ENABLE | DISTRIBUTED_MASTER) & ~SECOND_PASS & PHASE_0 & **~ap***.

NOTE—The decision to set DISTRIBUTED_ARBITER_REQUEST is subject to metastability. Modules shall not release **ar*** until adequate time has been allowed for the resolution of this condition.

DEPOSITION. A module may set DEPOSITION if ~CENTRAL_ARBITER & ~INIT & MASTER_ENABLE & ~BUS_HOLD & PHASE_3 & **ap*** if it needs to send an arbitration message or if it needs to obtain bus mastership and will use a higher arbitration number (**PR[7..0]**, **RR**, **GA[4..0]**) than that of the current winner. Modules shall maintain DEPOSITION set until PHASE_0.

NOTE—The decision to set DEPOSITION is subject to metastability. Modules shall not release **ap*** until adequate time has been allowed for the resolution of this condition.

NO_ARBITRATION_OWNER. Modules shall set no_ARBITRATION_OWNER when init and shall maintain it set until phase_5 & **~AC1**.

ARBITRATION_OWNER. A module shall set ARBITRATION_OWNER if ~INIT & (DISTRIBUTED_ARBITER_REQUEST & WINNER & ~SECOND_PASS_REQUIRED & **~AC1***) & PHASE_5 and maintain it set until INIT | (~DISTRIBUTED_ARBITER_REQUEST & ~SECOND_PASS_REQUIRED & **~AC1*** | DISTRIBUTED_ARBITER_REQUEST & ~WINNER & ~SECOND_PASS_REQUIRED & **~AC1***) & PHASE_5.

DISTRIBUTED_GRANT. A module shall set DISTRIBUTED_GRANT when DISTRIBUTED_ARBITER_REQUEST & (WINNER & ~SECOND_PASS_REQUIRED & **~AC1*** & PHASE_5 | ARBITRATION_OWNER). A module shall clear DISTRIBUTED_GRANT when ~DISTRIBUTED_ARBITER_REQUEST.

DISTRIBUTED_MASTER. A module shall set DISTRIBUTED_MASTER when DISTRIBUTED_GRANT & **~ET***. A module shall clear DISTRIBUTED_MASTER when ~DISTRIBUTED_ARBITER_REQUEST & **~et***.

TWO_PASS. A module shall set TWO_PASS when ~(**PR7** & **PR6** & **PR5** & **PR4** & **PR3** & **PR2** & **PR1**).

PR[7..0]. A module shall load an 8-bit priority value into **PR[7..0]** when ~SECOND_PASS & (PHASE_0 | PHASE_1) & DISTRIBUTED_ARBITER_REQUEST & **ar***.

ROUND_ROBIN. Modules shall either maintain a ROUND_ROBIN attribute for each priority class or if multiple priority classes are associated with a single ROUND_ROBIN attribute shall clear ROUND_ROBIN each time they changes the priority class associated with the ROUND_ROBIN attribute. Priority Class is defined to be a set of competition numbers that have identical priority fields and are discriminated by the Round Robin and Geographical Address fields.

A module shall set ROUND_ROBIN for a priority class if ~INIT & UNIQUENESS_GREATER & PHASE_5 & **~AC1*** and shall maintain it set until INIT | ~UNIQUENESS_GREATER & PHASE_5 & **~AC1***. A module may maintain ROUND_ROBIN set if BUSY_STATUS | WAIT_STATUS is received. A module may set ROUND_ROBIN if it has relinquished bus tenure early to a module of higher priority class.

GA[4..0]. A 5-bit value, **GA[4..0]**, shall be set to **GA[4..0]***.

UNIQUENESS_GREATER. Modules shall set UNIQUENESS_GREATER if PHASE_3 & ~DISTRIBUTED_MESSAGE & **AB7*** and the value on **AB[4..0]*** is greater than **GA[4..0]** and its priority

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

level is the same as that of the current winner. Modules shall maintain UNIQUENESS_GREATER until PHASE_0.

5.2.4 General arbitration and message attributes

COMPETITOR. A module shall set COMPETITOR if ~INIT & ~SECOND_PASS & (ARBITRATED_MESSAGE_REQUEST & ~ARBITRATED_MESSAGE_SENT | CENTRAL_MESSAGE_REQUEST & ~CENTRAL_MESSAGE_SENT | DISTRIBUTED_ARBITER_REQUEST & ~ARBITRATION_OWNER | DISTRIBUTED_MESSAGE_REQUEST & ~DISTRIBUTED_MESSAGE_SENT) & (PHASE_0 | PHASE_1) & **ar*** and shall maintain it set until INIT | (**AC0*** | ~SECOND_PASS_REQUIRED | ~WINNER) & PHASE_5.

CANCEL. Modules set CANCEL if phase_3 & (DEPOSITION | SECOND_PASS_REQUIRED & COMPETITOR | ARBITRATION_ERROR | DISTRIBUTED_MESSAGE) | PHASE_4 & ARBITRATION_OWNER & TRANSACTION_BEGIN. Modules shall clear CANCEL if PHASE_0.

CN7. A module shall set **cn7** when ARBITRATED_MESSAGE_REQUEST | CENTRAL_MESSAGE_REQUEST & SECOND_PASS | DISTRIBUTED_ARBITER_REQUEST & (~TWO_PASS | SECOND_PASS) | DISTRIBUTED_MESSAGE_REQUEST.

CN6. A module shall set **cn6** when ARBITRATED_MESSAGE_REQUEST & **AM6** | CENTRAL_MESSAGE_REQUEST & (~SECOND_PASS & **AM13** | SECOND_PASS & **AM6**) | DISTRIBUTED_ARBITER_REQUEST & ((~TWO_PASS | SECOND_PASS) & **PR0** | TWO_PASS & ~SECOND_PASS & **PR7**) | DISTRIBUTED_MESSAGE_REQUEST & (~SECOND_PASS | SECOND_PASS & **AM6**).

CN5. A module shall set **cn5** when ARBITRATED_MESSAGE_REQUEST & **AM5** | CENTRAL_MESSAGE_REQUEST & (~SECOND_PASS & **AM12** | SECOND_PASS & **AM5**) | DISTRIBUTED_ARBITER_REQUEST & ((~TWO_PASS | SECOND_PASS) & ROUND_ROBIN | TWO_PASS & ~SECOND_PASS & **PR6**) | DISTRIBUTED_MESSAGE_REQUEST & (~SECOND_PASS | SECOND_PASS & **AM5**).

CN4. A module shall set **cn4** when ARBITRATED_MESSAGE_REQUEST & **AM4** | CENTRAL_MESSAGE_REQUEST & (~SECOND_PASS & **AM11** | SECOND_PASS & **AM4**) | DISTRIBUTED_ARBITER_REQUEST & ((~TWO_PASS | SECOND_PASS) & **GA4** | TWO_PASS & ~SECOND_PASS & **PR5**) | DISTRIBUTED_MESSAGE_REQUEST & (~SECOND_PASS | SECOND_PASS & **AM4**).

CN3. A module shall set **cn3** when ARBITRATED_MESSAGE_REQUEST & **AM3** | CENTRAL_MESSAGE_REQUEST & (~SECOND_PASS & **AM10** | SECOND_PASS & **AM3**) | DISTRIBUTED_ARBITER_REQUEST & ((~TWO_PASS | SECOND_PASS) & **GA3** | TWO_PASS & ~SECOND_PASS & **PR4**) | DISTRIBUTED_MESSAGE_REQUEST & (~SECOND_PASS | SECOND_PASS & **AM3**).

CN2. A module shall set **cn2** when ARBITRATED_MESSAGE_REQUEST & **AM2** | CENTRAL_MESSAGE_REQUEST & (~SECOND_PASS & **AM9** | SECOND_PASS & **AM2**) | DISTRIBUTED_ARBITER_REQUEST & ((~TWO_PASS | SECOND_PASS) & **GA2** | TWO_PASS & ~SECOND_PASS & **PR3**) | DISTRIBUTED_MESSAGE_REQUEST & (~SECOND_PASS | SECOND_PASS & **AM2**).

CN1. A module shall set **cn1** when ARBITRATED_MESSAGE_REQUEST & **AM1** | CENTRAL_MESSAGE_REQUEST & (~SECOND_PASS & **AM8** | SECOND_PASS & **AM1**) | DISTRIBUTED_ARBITER_REQUEST & ((~TWO_PASS | SECOND_PASS) & **GA1** | TWO_PASS &

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

~SECOND_PASS & **PR2**) | DISTRIBUTED_MESSAGE_REQUEST & (~SECOND_PASS | SECOND_PASS & **AM1**).

CN0. A module shall set **cn0** when ARBITRATED_MESSAGE_REQUEST & **AM0** | CENTRAL_MESSAGE_REQUEST & (~SECOND_PASS & **AM7** | SECOND_PASS & **AM0**) | DISTRIBUTED_ARBITER_REQUEST & ((~TWO_PASS | SECOND_PASS) & **GA0** | TWO_PASS & ~SECOND_PASS & **PR1**) | DISTRIBUTED_MESSAGE_REQUEST & (~SECOND_PASS | SECOND_PASS & **AM0**).

CNP. A module shall set **cnp** if the number of **cn[7..0]** attributes it has set is even.

SECOND_PASS_REQUIRED. Modules shall set SECOND_PASS_REQUIRED if ~SECOND_PASS & PHASE_3 & (~**AB7*** | DISTRIBUTED_MESSAGE) and shall maintain it set until phase_0.

SECOND_PASS. Modules shall set SECOND_PASS if ~INIT & SECOND_PASS_REQUIRED & ~**AC0*** & PHASE_5 and shall maintain it set until INIT | (~SECOND_PASS_REQUIRED | **AC0***) & PHASE_5.

5.2.5 Arbitration timing attributes

PHASE_0. Modules shall set PHASE_0 and clear PHASE_5 if all required PHASE_5 activities are complete and PHASE_5 & (~**AQf** | **AP***). Modules shall set PHASE_0 if INIT.

PHASE_1. Modules shall set PHASE_1 and clear PHASE_0 if all required PHASE_0 activities are complete and PHASE_0 & (**ap*** | **AP***). Modules shall clear PHASE_1 if INIT.

PHASE_2. Modules shall set PHASE_2 and clear PHASE_1 if all required PHASE_1 activities are complete and PHASE_1 & (~**ARf** | **AQ***). Modules shall clear PHASE_2 if INIT.

PHASE_3. Modules shall set PHASE_3 and clear PHASE_2 if all required PHASE_2 activities are complete and PHASE_2 & (**aq*** | **AQ***). Modules shall clear PHASE_3 if INIT.

PHASE_4. Modules shall set PHASE_4 and clear PHASE_3 if all required PHASE_3 activities are complete and PHASE_3 &(~**APf** | **AR***). Modules shall clear PHASE_4 if INIT.

PHASE_5. Modules shall set PHASE_5 and clear PHASE_4 if all required PHASE_4 activities are complete and PHASE_4 & (**ar*** | **AR***). Modules shall clear PHASE_5 if INIT.

COMPETE. A module shall set COMPETE when COMPETITOR & PHASE_1 and maintain it set until PHASE_4.

SETTLED. Modules shall clear SETTLED if PHASE_0.

A COMPETITOR shall set SETTLED if a sufficient period of time $\mathbf{t_a}$ has passed since the beginning of PHASE_2 that the competitor is guaranteed to have won the competition | PHASE_3.

WINNER. A module shall set WINNER when SETTLED & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***) & (**cn5** | ~**AB5***) & (**cn4** | ~**AB4***) & (**cn3** | ~**AB3***) & (**cn2** | ~**AB2***) & (**cn1** | ~**AB1***) & (**cn0** | ~**AB0***) and maintain WINNER set until PHASE_0.

5.2.6 Arbitration error attributes

ARBITRATION_ERROR. Modules should set ARBITRATION_ERROR if ARBITRATION_TIMEOUT | ARBITRATION_PARITY_ERROR & PARITY_REPORT_ENABLE | ARBITRATION_COMPARISON_ERROR. Modules shall clear ARBITRATION_ERROR if PHASE_0.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

ARBITRATION_PARITY_ERROR. Modules should set ARBITRATION_PARITY_ERROR during PHASE_3 if the number of asserted **AB[ ]*** lines is even and **ABP*** is released or if the number of **AB[ ]*** lines is odd and **ABP*** is asserted. Modules shall maintain ARBITRATION_PARITY_ERROR set until PHASE_0.

ARBITRATION_COMPARISON_ERROR. Modules should set ARBITRATION_COMPARISON_ERROR if PHASE_3 & COMPETITOR & (**AB[ ]*** < **cn[ ]**).

ARBITRATION_TIMEOUT. A module should set ARBITRATION_TIMEOUT if (PHASE_2 remains true for 1–2 μs) | (PHASE_4 remains true for 1–2 μs) and maintain it set until PHASE_0.

5.2.7 Signal definition

5.2.7.1 Arbitration synchronization

5.2.7.1.1 AP*

Modules shall assert **ap*** when PHASE_0 & (ARBITRATED_MESSAGE_REQUEST & ~ARBITRATED_MESSAGE_SENT | CENTRAL_MESSAGE_REQUEST & ~CENTRAL_MESSAGE_SENT | DISTRIBUTED_ARBITER_REQUEST & ~ARBITRATION_OWNER | DISTRIBUTED_MESSAGE_REQUEST & ~DISTRIBUTED_MESSAGE_SENT) & ~BUS_HOLD | PHASE_1 and shall maintain **ap*** asserted until INIT | PHASE_3 & (CENTRAL_ARBITER | ~CENTRAL_ARBITER & (**ac1*** | **ac1*** | **~as***)) and the module has latched the winning arbitration number and checked it for errors.

5.2.7.1.2 AQ*

Modules shall assert **aq*** when PHASE_2 & (WINNER | ARBITRATION_TIMEOUT) | PHASE_3 and shall maintain **aq*** asserted until INIT | PHASE_5 and both **AC0*** and **AC1*** have been sampled.

5.2.7.1.3 AR*

Modules shall assert **ar*** when (~POWER_UP | BUS_IDLE_1US) & INIT | PHASE_4 & (CENTRAL_ARBITER | ~CENTRAL_ARBITER & (ARBITRATION_OWNER & TRANSACTION_COMPLETE | NO_ARBITRATION_OWNER & WINNER | ARBITRATION_TIMEOUT | **AC1*** | **ac1***)) | PHASE_5 and shall maintain it asserted until **ap*** | PHASE_1.

5.2.7.2 Arbitration condition

5.2.7.2.1 AC0*

Modules shall assert **ac0*** when PHASE_3 & ARBITRATION_ERROR. Modules shall maintain **ac0*** set until PHASE_0.

5.2.7.2.2 AC1*

A module shall assert **ac1*** when CANCEL and shall maintain it set until PHASE_0.

5.2.7.3 Arbitration bus

5.2.7.3.1 AB7*

A COMPETITOR shall assert **ab7*** when **cn7** & COMPETE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.7.3.2 AB6*

A COMPETITOR shall assert **ab6*** when **cn6** & COMPETE & (**cn7** | ~**AB7***).

5.2.7.3.3 AB5*

A COMPETITOR shall assert **ab5*** when **cn5** & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***).

5.2.7.3.4 AB4*

A COMPETITOR shall assert **ab4*** when **cn4** & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***) & (**cn5** | ~**AB5***).

5.2.7.3.5 AB3*

A COMPETITOR shall assert **ab3*** when **cn3** & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***) & (**cn5** | ~**AB5***) & (**cn4** | ~**AB4***).

5.2.7.3.6 AB2*

A COMPETITOR shall assert **ab2*** when **cn2** & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***) & (**cn5** | ~**AB5***) & (**cn4** | ~**AB4***) & (**cn3** | ~**AB3***).

5.2.7.3.7 AB1*

A COMPETITOR shall assert **ab1*** when **cn1** & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***) & (**cn5** | ~**AB5***) & (**cn4** | ~**AB4***) & (**cn3** | ~**AB3***) & (**cn2** | ~**AB2***).

5.2.7.3.8 AB0*

A COMPETITOR shall assert **ab0*** when **cn0** & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***) & (**cn5** | ~**AB5***) & (**cn4** | ~**AB4***) & (**cn3** | ~**AB3***) & (**cn2** | ~**AB2***) & (**cn1** | ~**AB1***).

5.2.7.3.9 ABP*

A COMPETITOR shall assert **abp*** when **cnp** & COMPETE & (**cn7** | ~**AB7***) & (**cn6** | ~**AB6***) & (**cn5** | ~**AB5***) & (**cn4** | ~**AB4***) & (**cn3** | ~**AB3***) & (**cn2** | ~**AB2***) & (**cn1** | ~**AB1***) & (**cn0** | ~**AB0***).

5.2.8 Protocol definition—distributed arbitration and messages

5.2.8.1 Phase 0—idle

A module shall clear its DEPOSITION, UNIQUENESS_GREATER, CANCEL, SETTLED, SECOND_PASS_REQUIRED, WINNER, ARBITRATION_ERROR, ARBITRATION_PARITY_ERROR, and ARBITRATION_TIMEOUT attributes, if set, and release its **ac0*** and **ac1*** signals, if asserted.

A module shall clear DISTRIBUTED_MESSAGE if ~SECOND_PASS.

An ARBITRATION_OWNER may set DISTRIBUTED_GRANT if DISTRIBUTED_ARBITER_REQUEST.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

If DISTRIBUTED_ARBITER_REQUEST & ~ARBITRATION_OWNER | DISTRIBUTED_MESSAGE_REQUEST & ~DISTRIBUTED_MESSAGE_SENT a module shall assert **ap***.

5.2.8.2 Phase 1—decision

If (DISTRIBUTED_ARBITER_REQUEST & ~ARBITRATION_OWNER | DISTRIBUTED_MESSAGE_REQUEST & ~DISTRIBUTED_MESSAGE_SENT) & ~SECOND_PASS a module shall

1) Ensure that any metastability conditions have been adequately resolved.
2) Set COMPETITOR.
3) Update **PR[7..0]**, ROUND_ROBIN, **GA[4..0]**, TWO_PASS, and **AM[7..0]**.

If DISTRIBUTED_ARBITER_REQUEST & ~ARBITRATION_OWNER | DISTRIBUTED_MESSAGE_REQUEST & ~DISTRIBUTED_MESSAGE_SENT a module shall

1) Update **cn[7..0]** and **cnp**.
2) Set COMPETE.
3) Ensure that enough time has elapsed for the most significant bit of **cn[7..0]** to have been enabled on the corresponding **AB[7..0]*** line.

All modules shall assert **ap***, if not already asserted.

All modules shall then release **ar***.

5.2.8.3 Phase 2—competition

If a module remains in PHASE_2 for more than 1 to 2 μs, it shall set ARBITRATION_TIMEOUT and ARBITRATION_ERROR, and assert **aq***.

A COMPETITOR shall

1) Begin timing a period $\mathbf{t_a}$ as specified in 5.2.5.
2) If SETTLED and WINNER, ensure that **abp*** is valid, and then assert **aq***.

5.2.8.4 Phase 3—error check

All modules shall assert **aq***, if not already asserted.

All modules shall

1) Latch the values on **AB[7..0]*** and **ABP***.
2) Set ARBITRATION_PARITY_ERROR and ARBITRATION_ERROR if parity is incorrect on **AB[7..0]*** and **ABP***.
3) Set DISTRIBUTED_MESSAGE if ~SECOND_PASS & **AB7*** & **AB6*** & **AB5*** & **AB4*** & **AB3*** & **AB2*** & **AB1*** & **AB0***.
4) Set UNIQUENESS_GREATER if specified in 5.2.3.
5) Set SECOND_PASS_REQUIRED if ~**AB7*** | DISTRIBUTED_MESSAGE.
6) Set CANCEL if ARBITRATION_ERROR | DISTRIBUTED_MESSAGE | SECOND_PASS_REQUIRED & COMPETITOR.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If DEPOSITION, a module shall

1) Ensure that any metastability conditions have been adequately resolved.
2) Set CANCEL.

All modules shall

1) If ARBITRATION_ERROR, assert **ac0***.
2) If CANCEL, assert **ac1***.

If **AC1*** | ~**AS***, all modules shall then release **ap***.

5.2.8.5 Phase 4—release mastership

If a module remains in PHASE_4 for more than 1 to 2 μs, it shall set ARBITRATION_TIMEOUT and ARBITRATION_ERROR, and assert **ar***.

A COMPETITOR shall

1) Clear COMPETE.
2) If WINNER & NO_ARBITRATION_OWNER, assert **ar***.

An ARBITRATION_OWNER shall

1) If TRANSACTION_BEGIN, set CANCEL, assert **ac1***, and then assert **ar***.
2) If TRANSACTION_COMPLETE, assert **ar***.

All modules shall assert **ar*** if **AC1***.

5.2.8.6 Phase 5—transfer tenure

All modules shall assert **ar***, if not already asserted.

A module shall release its **ab[ ]*** and **abp*** signals, if asserted.

A module shall

1) Evaluate the values on **AC0*** and **AC1***.
2) Update ROUND_ROBIN as specified in 5.2.1.4.
3) Set SECOND_PASS if SECOND_PASS_REQUIRED & ~**AC0***.
4) Clear NO_ARBITRATION_OWNER if ~**AC1***.

A COMPETITOR shall

1) Set ARBITRATION_OWNER and DISTRIBUTED_GRANT if DISTRIBUTED_ARBITER_REQUEST & WINNER & ~SECOND_PASS_REQUIRED & ~**AC1***.
2) Set DISTRIBUTED_MESSAGE_SENT if DISTRIBUTED_MESSAGE_REQUEST & WINNER & ~SECOND_PASS_REQUIRED & ~**AC0***.
3) Clear COMPETITOR if ~WINNER | ~SECOND_PASS_REQUIRED | **AC0***.

An ARBITRATION_OWNER shall clear ARBITRATION_OWNER if ~DISTRIBUTED_ARBITER_REQUEST & ~SECOND_PASS_REQUIRED & ~**AC1*** | DISTRIBUTED_ARBITER_REQUEST & ~WINNER & ~SECOND_PASS_REQUIRED & ~**AC1***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

All modules shall then release **aq***.

5.2.9 Protocol definition—central arbiter messages

5.2.9.1 Phase 0—idle

A module shall clear its CANCEL, SETTLED, SECOND_PASS_REQUIRED, WINNER, ARBITRATION_ERROR, ARBITRATION_PARITY_ERROR, and ARBITRATION_TIMEOUT attributes, if set, and release its **ac0*** and **ac1*** signals, if asserted.

If ARBITRATED_MESSAGE_REQUEST & ~ARBITRATED_MESSAGE_SENT | CENTRAL_MESSAGE_REQUEST & ~CENTRAL_MESSAGE_SENT a module shall assert **ap***.

5.2.9.2 Phase 1—decision

If (ARBITRATED_MESSAGE_REQUEST & ~ARBITRATED_MESSAGE_SENT | CENTRAL_MESSAGE_REQUEST & ~CENTRAL_MESSAGE_SENT) & ~SECOND_PASS a module shall

1) Ensure that any metastability conditions have been adequately resolved.
2) Set COMPETITOR.
3) Update **PR[7..0]**, **RQ**, and **AM[13..0]**.

If ARBITRATED_MESSAGE_REQUEST & ~ARBITRATED_MESSAGE_SENT | CENTRAL_MESSAGE_REQUEST & ~CENTRAL_MESSAGE_SENT) a module shall

1) Update **cn[7..0]**, and **cnp**.
2) Set COMPETE.
3) Ensure that enough time has elapsed for the most significant bit of **cn[7..0]** to have been enabled on the corresponding **AB[7..0]*** line.

All modules shall assert **ap***, if not already asserted.

All modules shall then release **ar***.

5.2.9.3 Phase 2—competition

If a module remains in PHASE_2 for more than 1 to 2 μs, it shall set ARBITRATION_TIMEOUT and ARBITRATION_ERROR, and assert **aq***.

A COMPETITOR shall

1) Begin timing a period $\mathbf{t_a}$ as specified in 5.2.5.
2) If SETTLED and WINNER, ensure that **abp*** is valid, and then assert **aq***.

5.2.9.4 Phase 3—error check

All modules shall assert **aq***, if not already asserted.

All modules shall

1) Latch the values on **AB[7..0]*** and **ABP***.
2) Set ARBITRATION_PARITY_ERROR and ARBITRATION_ERROR if parity is incorrect on **AB[7..0]*** and **ABP***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3) Set SECOND_PASS_REQUIRED if ~**AB7***.
4) Set CANCEL if ARBITRATION_ERROR | SECOND_PASS_REQUIRED & COMPETITOR.

All modules shall

1) If ARBITRATION_ERROR, assert **ac0***.
2) If CANCEL, assert **ac1***.

All modules shall then release **ap***.

5.2.9.5 Phase 4—release mastership

If a module remains in PHASE_4 for more than 1 to 2 μs, it shall set ARBITRATION_TIMEOUT and ARBITRATION_ERROR, and assert **ar***.

A COMPETITOR shall clear COMPETE.

All modules shall assert **ar***.

5.2.9.6 Phase 5—transfer tenure

All modules shall assert **ar***, if not already asserted.

A module shall release its **ab[ ]*** and **abp*** signals, if asserted.

A module shall

1) Evaluate the value on **AC0***.
2) Set SECOND_PASS if SECOND_PASS_REQUIRED & ~**AC0***.

A COMPETITOR shall

1) Set ARBITRATED_MESSAGE_SENT if ARBITRATED_MESSAGE_REQUEST & WINNER & ~**AC0***.
2) Set CENTRAL_MESSAGE_SENT if CENTRAL_MESSAGE_REQUEST & WINNER & ~**AC0*** & ~SECOND_PASS_REQUIRED.
3) Clear COMPETITOR if ~WINNER | ~SECOND_PASS_REQUIRED | **AC0***.

All modules shall then release **aq***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6. Parallel protocol

6.1 Description

This clause describes and specifies how data is transferred using Futurebus+.

6.1.1 Mastership

Having gained control of the bus through the arbitration process, a master may use the data transfer bus to conduct *bus transactions* between itself and one or more slaves. The master's control of the bus is known as its *bus tenure*. It should be noted that during a master's bus tenure, a master also performs all the functions of a slave.

There is no specified limit to the number or duration of transactions the master may conduct before relinquishing control of the bus. Once a master has gained control of the bus, it cannot, under normal circumstances, be removed. It is the responsibility of the system integrator to restrict a master's bus tenure in order to optimize the overall performance of the system. Profiles or other higher standards will generally specify a maximum tenure.

Some types of bus transactions are part of higher level system transactions. These system transactions, which involve multiple bus transactions, are referred to as *split transactions*. A complete uncached split read or write transaction requires two bus transactions. A complete split cached transaction may involve any number of bus transactions.

6.1.2 Transactions

A module uses the data transfer bus to conduct a system transaction. A system transaction consists of a request followed by a response. There are two ways to conduct a system transaction:

1) A *connected transaction* is used to conduct a module's request and response in a single bus transaction.
2) A *split transaction* is used to split a module's request and response phases into separate bus tenures and transactions. The module servicing the request responds by becoming a master, addressing the requester, and transmitting a response.

6.1.3 Bus transaction phases

A master may conduct bus transactions during its bus tenure. Each bus transaction comprises up to three phases:

1) A *connection* phase, where the master selects and establishes a connection to the desired slave(s).
2) An optional *data transfer* phase, where data are transferred between the master and the connected slave(s).
3) A *disconnection* phase, where the master terminates the transaction and disconnects from the slave(s).

All bus transactions are required to have a connection and a disconnection phase. The data transfer phase is not included in some types of transactions. Transactions without a data transfer phase are generally referred to as *address only* transactions. This type of transaction is used in various ways to implement split transactions. Address only transactions may also be used for system event messages that, for example, may set a simple primitive variable, such as a targeted interrupt within a processor module. The destination and action of the command are implied within the address and command.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A transaction begins with the master establishing a connection between itself and one or more slaves during the connection phase. The data transfer phase begins when the connection is established and ends when it is broken. The master may transfer data with the connected slave(s) during this phase. The connection is broken during the disconnection phase.

During the data transfer phase, the master may conduct *data transfers* using either the *compelled* data transfer mode, or the *packet* data transfer mode.

The master may conduct one of the following bus transactions:

1) An *address only transaction* consisting of an address transfer followed by the disconnection phase. There is no data transfer phase, and hence no data is transferred with the exception of the information contained within the connection and disconnection phases.
2) A *compelled data transaction* consisting of an address transfer followed by a block of one or more data transfers to one or more contiguous addresses beginning with the address in the address transfer. Each data transfer is controlled by a fully compelled handshake between the master and slave(s).
3) A *packet data transfer* consisting of an address transfer followed by a fixed length block of data from a set of contiguous addresses beginning with the address in the address transfer. Data transfers within a packet use a noncompelled protocol. Each complete packet transfer is controlled by a fully compelled handshake between the master and the slave(s).

6.1.4 Data transfer protocols

The compelled protocol is a technology-independent transfer mechanism. This type of handshake is referred to as *compelled* because the slave is compelled to provide a response before the master proceeds. The second, referred to as the packet protocol, is a technology-dependent noncompelled transfer mechanism at the individual transfer level and a compelled protocol at the packet level. The packet protocol is optimized for high-speed block transfers and does not include many of the features available in the compelled protocols. Transactions and operations not specifically identified as valid are prohibited.

Which protocol is to be used is negotiated during the connection phase of the transaction. Any participant may force the transfer to be performed with the compelled protocol.

The participants in a packet transaction must all operate at the same transfer rate. All of the modules capable of operating with the packet mode perform a system-wide negotiation during initialization to configure the two standard transfer rates for that system. Each of these modules provide a register from which other modules may read its supported transfer rates. (See clause 7 for details.) The monarch processor reads the supported transfer rates from all the other modules, and sets the low transmit speed equal to the lowest speed of all the packet modules on the bus. The high transmit speed is set equal to the highest speed at which an implementor chosen subset of the modules can operate. Determination of which of two transmit speeds is to be used during a particular transaction is negotiated during the connection phase of the transaction. Any participating module may force the transfer to be performed using the lower speed.

Because multiple data transfers occur after a single data handshake in packet mode, participating modules must know in advance how many transfers to perform. Each module capable of packet operation provides a register from which other modules may read its supported packet lengths. (See clause 7 for details.) Only packet transfer lengths 2, 4, 8, 16, 32, and 64 are permitted. The master specifies one of these standard packet lengths during the data phase. In compelled transactions, data transfers may be any length within the constraints of the transaction timeout when using the *unrestricted* data transfer length, unless restricted by a profile.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.5 Broadcast handshake

Many slaves will have a unique address range in a system, and transactions to these slaves will normally use a *single slave* handshake. Some slaves may share areas of the physical address space, and thus, when a master accesses an address in one of these areas, all the slaves respond together. Such a transaction is known as a *broadcast* or *broadcall* transaction and the participating modules use the broadcast handshake. The participating slaves in a transaction may cause the transaction to use the broadcast handshake. The broadcast handshake is also used frequently in cache coherent, shared memory systems. This occurs when a cache detects that data in which it is interested is being transferred on the bus. The cache induces the broadcast handshake and reads the data. In a caching environment this broadcast operation is referred to as *snarfing*.

6.1.6 Intervention

Another variety of transaction that is used in a caching environment occurs when a master selects a memory module to read a block of data, and a cache on another module sees that it has a more up-to-date copy. The cache then intervenes in the transaction by diverting the memory module, and provides the data to the master and to the memory module. This is referred to as *intervention*.

Only one slave intervenes in a transaction. Clause 8 explains how cache line attributes are used to guarantee one intervener.

6.1.7 Cache line status

In a caching environment, modules dynamically keep track of the attributes of cache lines. They do this by monitoring bus signals during bus transactions. This is explained in more detail in clause 8.

6.1.8 Split transactions

Most connected transactions may be converted to split transactions. Split transactions allow modules with long data latency (e.g., access time) to use the bus resources in a more efficient way. A complete uncached split read or write transaction requires two bus transactions. A complete split cached transaction may involve any number of bus transactions. First, a bus transaction is initiated by the requesting module which is split by another module. The requesting module then terminates its bus tenure. When the responding module is able to provide the requested data or response, it arbitrates to become master of the bus. When it becomes master of the bus, it transfers the data to the requesting module. Therefore, data is always transferred from master to slave in split transactions; i.e., only write transactions are used.

Split transactions may require additional information beyond that necessary for connected transactions. For this purpose, facilities are provided to transfer the requestor's global identification, the requestor's original priority, and the responder's status in the disconnection phase.

If a read transaction is initiated and one of the participating slaves requires the transaction to be split, the connected transaction will be converted to a bus transaction that will request the desired data. After the master (the *requestor*) that requested the data ends its bus tenure, the module (the *responder*) with the requested data acquires the bus and transmits the data to the requestor.

A split write transaction consists of a write transaction followed by an address only bus transaction initiated by the responder that acknowledges the completion of the write.

Modules are not required to provide split response capability.

The priority management of split transactions is user defined and is not specified by this International Standard.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.9 Locked transactions

The locked transactions, as implemented by Futurebus+, may be divided into two categories, simple transactions and locked commands. Simple locked transactions, in this context, refer to those locked transactions that are not one of the defined set of locked commands. The locked commands are those defined during the data phase to be one of Mask and Swap, Compare and Swap or Fetch and Add, either Big or Little.

Locks occur within a system for a number of reasons, although these may be condensed roughly into two categories as follows. They are either involved with flags to ensure single access to a resource or program thread, or else they are involved with indivisible access to a data item, e.g., the update of a page table descriptor in an MMU table. The latter is associated with a lock on a single action, i.e., the read and modification of a variable, whereas the former implies a longer term lock where a variable is used as a flag or gate to ensure exclusive access to some larger structure or resource until such time as the requirement for exclusive access to that entity is complete.

To date, processors have not offered transaction or instruction types that allow such a differentiation to be made, with both types of action being performed using a simple set of protocols from the processor. The use of simple locked transactions on Futurebus+ maps to the transactions commonly implemented within processors and has some implications with respect to efficiency and performance, which must be fully understood to allow a correct choice to be made.

If a locked sequence of read and write transactions are performed on Futurebus+, then the bus, and the associated resource, becomes locked and unavailable for other use until that indivisible sequence is complete.

Where a simple read locked transaction is performed to a location that is accessed remotely across a bus bridge or other hierarchical link, there is an indeterminate penalty with respect to access time. This is undesirable as it may serve as an impediment to progress. Although the specification is unclear on what may be permitted, it may be stated categorically that such a transaction may not be split without destroying the indivisible nature of the intended sequence.

As a consequence of the dangers with respect to forward progress and indivisibility, there is a good argument to prohibit this type of transaction on Futurebus+, especially since this type of sequence may be performed quite safely within a cache coherent environment with no equivalent impact on system performance. Good practice dictates that such locked sequences should be performed without access to Futurebus+, either into local memory or else by the use of cache coherent techniques.

To ease the situation with respect to flags and gates, Futurebus+ defines a set of locked commands. This type of transaction is entirely suitable for the flags and gates required in a multiprocessor environment. The use of these commands is to be thoroughly encouraged within profiles. These transactions should be used wherever appropriate instead of the simple locked sequences.

Simple indivisible access to a variable should be performed without locked transactions on Futurebus+. This may be achieved by access to variables in local memory, or else in a cache coherent environment, where a locked read should generate a Read Modified transaction on Futurebus+. Permission to modify must be granted before that initial read may complete and indivisibility may be preserved by refusing to relinquish that privilege until the locked sequence is complete. With this technique, neither the bus nor any other external resource is tied up by the requirement to perform an indivisible sequence.

This technique should not be used for a spin lock, as it may result in excessive and unnecessary bus transactions. A more appropriate technique in a cache coherent environment, if the locked commands are not used, is to spin on the lock variable with a simple test instruction, moving onto the indivisible test and set instruction only when the variable is observed to be clear and available.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.10 Lock commands

The simple locking scheme described in 6.1.9 is not able to guarantee indivisible transactions efficiently in a split transaction environment. For example, for the typical processor generated semaphore of a read followed by a write, the master might simply make a read request followed by a write request with its accompanying data. The responder would return the requested read data as the first response, and a write acknowledge as the second response. Besides being inefficient, there is no guarantee the responder did not receive a request from another party between the two semaphore requests. If this occurred, the semaphore would not be atomic. The lock protocols might require the bus to prevent another request from being issued, but this would defeat the purpose of using split transactions. Because of this, facilities are provided to allow modules to perform lock operations with a single request referred to as a *lock command.* These lock commands may be used in either split or connected write transactions. In general, these lock commands will only be used in systems in which split transactions are implemented. The specified lock commands are *mask and swap, fetch and add,* and *compare and swap.* Bits in the module logical capability register indicate whether or not a module supports one or more lock commands.

The data manipulation operations involved in the mask-and-swap and compare-and-swap lock commands use logic for each bit in the operand's width that is independent of every other bit. Therefore these operations are independent of byte significance and the results will be identical for operations with either big or little endian ordering.

The fetch-and-add operation adds two operands to obtain the result. The adder logic must deal with carries between the bits. Bits within a byte always have a constant ordering, and carries always propagate in the same direction. Between bytes the carry propagation is dependent on the byte ordering of the operands. When the operands have little endian byte ordering, the carries propagate from **AD[7..0]*** to **AD[15..8]***, **AD[15..8]*** to **AD[23..16]***, etc. When the operands have big endian byte ordering, the carries for a 32-bit word propagate from **AD[31..24]*** to **AD[23..16]***, **AD[23..16]*** to **AD[15..8]***, etc. To support these different carry propagation directions there are two versions of fetch-and-add:

1) Fetch-and-add-big for big endian operand byte ordering
2) Fetch-and-add-little for little endian operand byte ordering

Locked read transactions are not usually used when split transactions can occur. If a locked read transaction is split, the lock only ensures that the block of data supplied by the responder was read internally with an atomic operation.

6.1.10.1 Mask and swap lock command

The connected mask and swap locked command, as shown in figure 14, consists of the following steps:

1) The master performs a write locked transaction, specifying the mask and swap command in the data phase, to transfer a mask operand to a slave.
2) Using the same transaction as step 1), the master transfers a swap operand to a slave. The master ends the transaction with a disconnection lock command. The slave does the following logical operation: MASK_SWAP_OPERAND = (MASK_OPERAND & SWAP_OPERAND) | (~MASK_OPERAND & ORIGINAL_DATA), and store it in a buffer.
3) Within the same bus tenure as steps 1) and 2) and before any other transactions, the master then performs a read locked transaction to retrieve the data contained in the addressed location prior to the write transaction from the slave.
4) The slave puts the masked swap operand in the place of the data contained in the addressed location prior to the write transaction.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

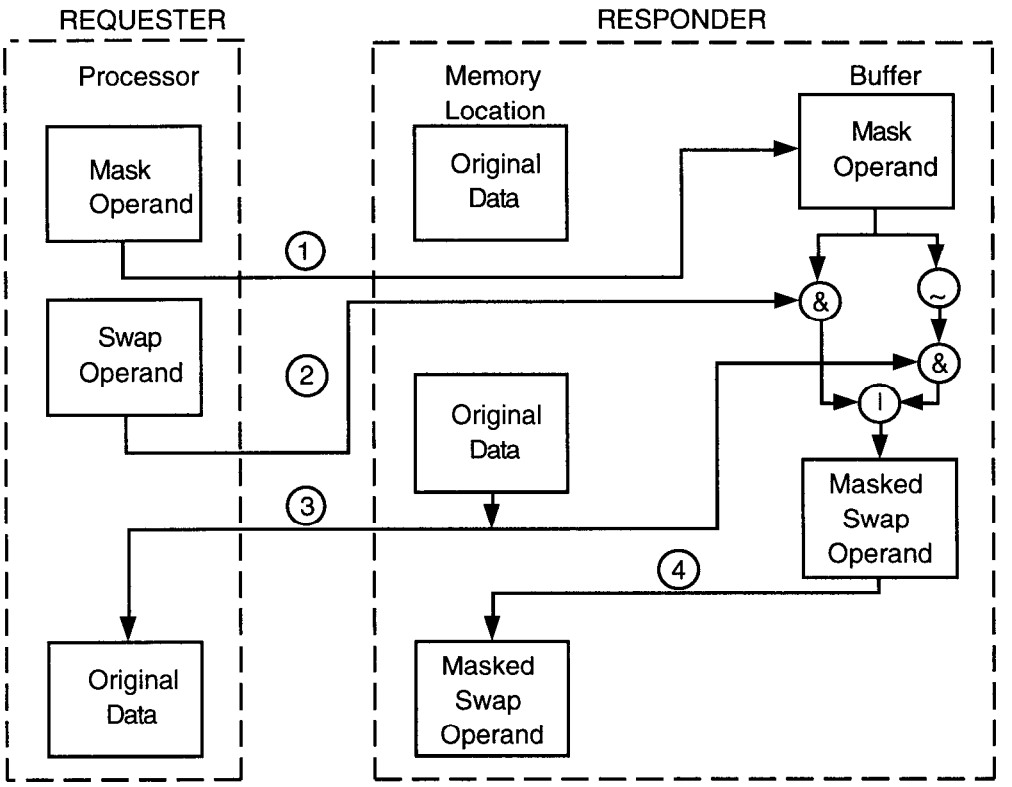


Figure 14—Mask and swap lock command

If, during the initial write transaction, the slave notifies the master that it desires the transaction to become a split transaction, the write transaction continues as in the connected case, after which the requesting module ends its bus tenure. When the responding module is ready it arbitrates for the bus and performs a read response transaction to return the data contained in the addressed location prior to the write transaction to the requesting module. The responder then puts the result of the mask and swap operation in place of the data contained in the addressed location prior to the write transaction.

6.1.10.2 Fetch and add lock command

In a connected fetch and add locked operation, as shown in figure 15, the following occurs:

1) The master first performs a write locked transaction, specifying the fetch and add lock command in the data phase, to transfer an add operand to the slave. The master ends the transaction with a disconnection lock command.
2) The slave adds the add operand to the data contained in the addressed location prior to the write transaction.
3) Within the same bus tenure as step 1) and before any other transactions, the master then performs a read locked transaction to retrieve the data contained in the addressed location prior to the write transaction from the slave.
4) The slave then puts the result into the addressed location.

If during the initial write transaction the slave notifies the master that it desires the transaction to become a split transaction, the write transaction continues as in the connected case after which the requesting module ends its bus tenure. When the responding module is ready, it arbitrates for the bus and performs a read response transaction to return the data contained in the addressed location prior to the write transaction to the requesting module. The responder adds the add operand to the data contained in the addressed location prior

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

to the write transaction and puts the result in place of the data contained in the addressed location prior to the write transaction.

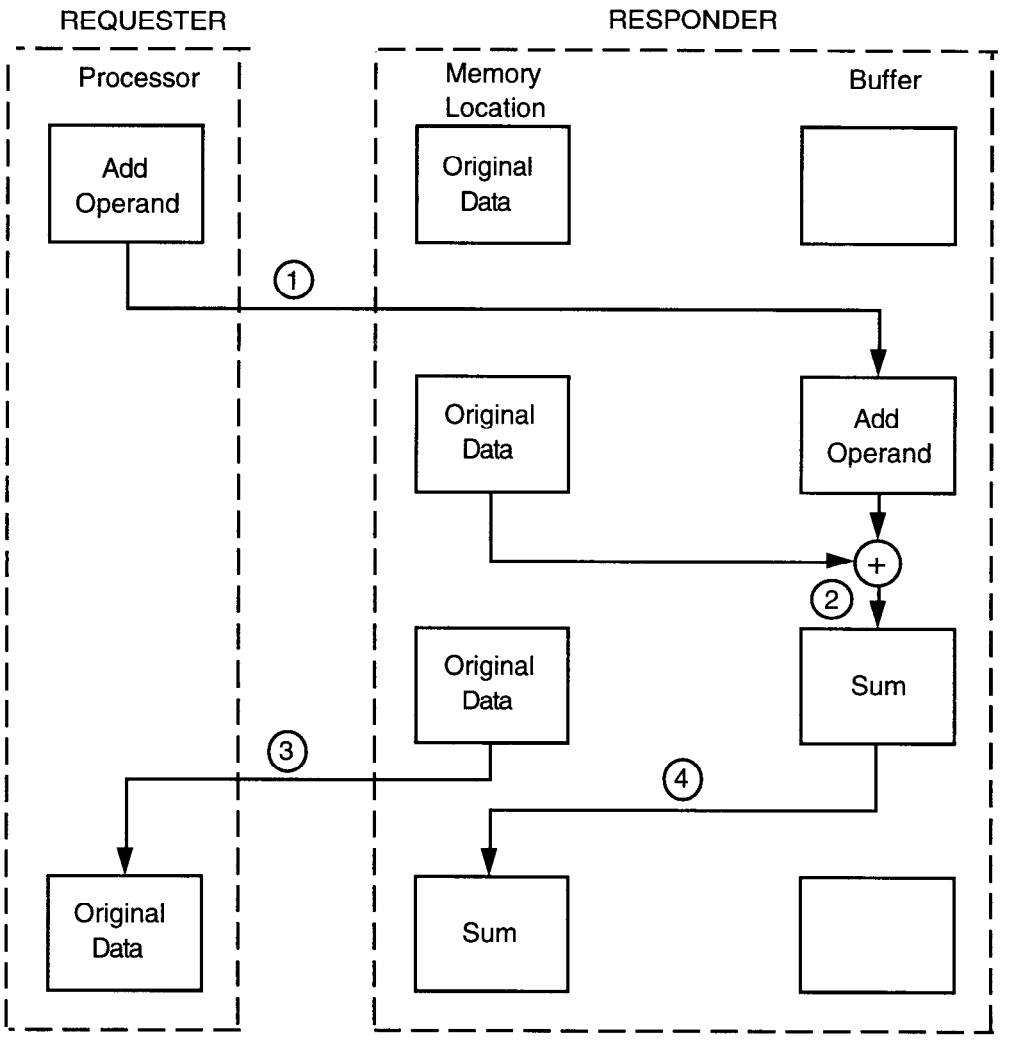


Figure 15—Fetch and add lock command

6.1.10.3 Compare and swap lock command

In a connected compare and swap locked operation, as shown in figure 16, the following occurs:

1) The master starts a write locked transaction, specifying the compare and swap lock command in the data phase, and then transfers a compare operand to the slave.
2) Using the same transaction as step 1), the master transfers a swap operand to the slave. The master ends the transaction with a disconnection lock command.
3) Within the same bus tenure as steps 1) and 2) and before any other transactions, the master then performs a read locked transaction to retrieve the data contained in the addressed location prior to the write transaction from the slave.
4) The slave then compares the compare operand with the data contained in the addressed location prior to the write transaction and if they are equal puts the swap operand in place of the data contained in the addressed location prior to the write transaction.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

RESULT = (ORIGINAL_DATA == COMPARE_OPERAND)?
SWAP_OPERAND: ORIGINAL_DATA.

If during the initial write transaction the slave notifies the master that it desires the transaction to become a split transaction, the write transaction continues as in the connected case after which the requesting module ends its bus tenure. When the responding module is ready, it arbitrates for the bus and performs a read response transaction to return the data contained in the addressed location prior to the write transaction to the requesting module. The responder then compares the compare operand with the data contained in the addressed location prior to the write transaction and, if they are equal, puts the swap operand in place of the data contained in the addressed location prior to the write transaction.

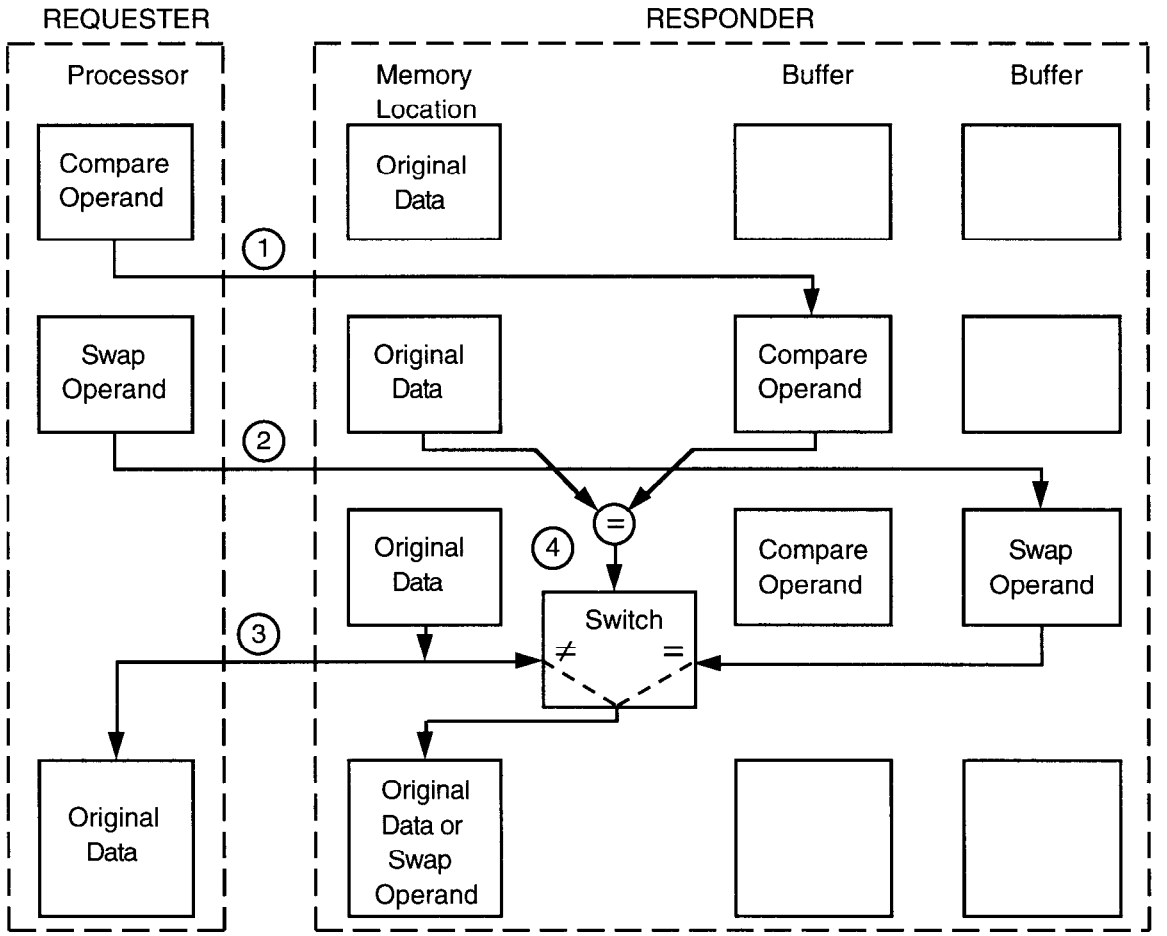


Figure 16—Compare and swap lock command

6.1.11 Busy

If an addressed module is unable to respond with a particular resource at the time it is requested to do so, it notifies the master it is *busy*. This could happen if the resource was currently being accessed by another port. If the master is performing locked transactions on multiple slaves and it receives busy status from any of them, it must unlock all the slaves and try the operation again later. This is to avoid deadlock problems that may occur if two modules on different ports are each waiting for resources that the other module has locked.

Transactions terminated with busy do not change the system state unless the busy retry count has been exceeded.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A master should provide a forward progress mechanism such as a time delay before trying to re-access this address. It is recommended that the duration of the time delay be varied during each subsequent retry to avoid live lock problems. Higher level specifications may specify this delay.

Forward progress is guaranteed in a system that does not exhibit deadlock or starvation. Starvation occurs when one or more modules perform no useful work for an indefinite period of time. Deadlock occurs when actors are waiting on actions that can only be performed by those waiting. Starvation may result from live-lock, a situation in which some modules acquire and release resources in a way that none is making progress.

While it is possible to build a Futurebus+ system guaranteeing no deadlock, designs using multiple buses may be vulnerable to livelock, and should be designed very carefully to minimize the probabilities of events that might result in such behavior. Most situations that could result in livelock result from the use of busy.

6.1.12 Wait

If a module is unable to respond with a particular resource at the time it is requested to do so, and if the module will later generate an externally visible event clearing the condition, the module notifies the master to *wait*. The externally visible event will normally be specified in a higher level standard. Transactions terminated with wait do not change the system state.

For example, the cache coherence protocol requires that, at most, one split request be outstanding for one address location on a given bus segment. A splitting bus bridge uses wait to delay a second request to the same location; the wait condition is cleared by the response transaction to the original request.

The split/wait handshake implements a queue where the head of the queue is the module which has its request accepted and the rest of the queue are the modules which have their requests waited. The order of service in the rest of the queue is determined by the bus arbitration process when the wait is satisfied. For real-time or priority based systems, the slave should increase the priority of the accepted request to the greater of the accepted request priority or the waited request's priority. This is referred to as priority inheritance.

A master should provide a timeout to signal an exception if the response does not arrive.

6.1.13 Extended bus width

Modules may be designed to support up to four data path widths. These are 32 bits, 64 bits, 128 bits, and 256 bits. The master specifies the bus width during the connection phase for a particular transaction.

If a module cannot support the data path specified in the connection phase, it asserts beat error status. See clause 7 for details of compatibility management through the interrogation of the error status of a module.

6.1.14 Extended address

Modules may be designed to support both 32-bit and 64-bit address path widths. The master specifies the address width during the connection phase for a particular transaction. Modules that cannot support a 64-bit address do not respond to such transactions. The 32-bit address space is a proper subset of the 64-bit address space. Transactions to the 32-bit address space should always use 32-bit address transactions in the interest of interoperability.

6.1.15 Module capabilities

The master and the participating slaves resolve any differences in their capabilities during the connection phase. The master specifies the type of transaction it wishes to perform. Participating modules that cannot

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

perform the specified transaction either assert an error status signal, or if possible, convert the transaction into one it is capable of.

6.1.16 Transactions

The uncached transactions specified in this International Standard are as follows:

— Read unlocked
— Write unlocked
— Address only unlocked
— Read locked
— Write locked
— Address only locked
— Read partial
— Write partial
— Read partial locked
— Write partial locked
— Write response
— Read response
— Write no acknowledge

The cached transactions specified in this International Standard are as follows:

— Read invalid
— Write invalid
— Read Shared
— Copyback
— Read modified
— Invalidate
— Shared response
— Modified response

6.1.16.1 Read unlocked

The *read unlocked* transaction is used to transfer data from one or more slaves to the master. The read unlocked transaction may be performed using either the compelled or the packet protocols. Slaves may convert a connected transaction into a split transaction in which the read transaction becomes an address only read request transaction. Slaves may induce broadcast and convert the transaction into a multiparty transaction. One of the slaves may induce intervention to supply the data instead of the selected slave.

6.1.16.2 Write unlocked

The *write unlocked* transaction is used to transfer data to one or more slaves from the master. This transaction may be performed using either the compelled or the packet protocols. Slaves may convert the transaction into a split transaction. Slaves may induce broadcast and convert the transaction into a multiparty transaction.

6.1.16.3 Address only unlocked

The *address only unlocked* transaction is used for system event messages. No data are transferred. This transaction can not be distinguished from write unlocked transaction during the connection phase. Modules determine it is an address only transaction by the absence of a data phase.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.16.4 Read locked

The *read locked* transaction is the same as the read unlocked transaction described in 6.1.16.1 except that it causes the participating slaves to lock any lockable addressed resources and by subsequent disconnection command maintains this locked condition past the end of the current transaction. This locked condition is maintained until an unlocked transaction or an end of tenure occurs.

6.1.16.5 Write locked

The *write locked* transaction is the same as the write unlocked transaction described in 6.1.16.2 except that it causes the participating slaves to lock any lockable addressed resources and by subsequent disconnection command maintains this locked condition past the end of the current transaction. This locked condition is maintained until an unlocked transaction or an end of tenure occurs.

6.1.16.6 Address only locked

The *address only locked* transaction is the same as the address only unlocked transaction described in 6.1.16.3 except that it causes the participating slaves to lock any lockable addressed resources and by subsequent disconnection command maintains this locked condition past the end of the current transaction. This locked condition is maintained until an unlocked transaction or an end of tenure occurs.

6.1.16.7 Read partial

The *read partial* transaction is used to transfer data from one or more slaves to the master. The master specifies that it will only evaluate some of the bytes within the specified data-path-width which are transferred. Slaves may convert the transaction into a split transaction. Slaves may induce broadcast and convert the transaction into a multiparty transaction.

6.1.16.8 Write partial

The *write partial* transaction is used to transfer data to one or more slaves. The master specifies to the slave which of the bytes transferred the slave should evaluate. Slaves may convert the transaction into a split transaction. Slaves may induce broadcast and convert the transaction into a multiparty transaction.

6.1.16.9 Read partial locked

The *read partial locked* transaction is the same as the read partial transaction described in 6.1.16.7 except that it causes the participating slaves to lock any lockable addressed resources and, by subsequent disconnection command, maintain this locked condition past the end of the current transaction.

6.1.16.10 Write partial locked

The *write partial locked* transaction is the same as the write partial transaction described in 6.1.16.8 except that it causes the participating modules to lock any lockable addressed resources and by subsequent disconnection command maintain this locked condition past the end of the current transaction.

6.1.16.11 Write response

A split *write response* consists of an address only transaction. The responding module transmits the requester's global identification during the connection phase. The requestor's global identification is transmitted in the connection phase, and status is transmitted in the disconnection phase. A write response transaction is an indication to the requesting module that a previously requested write transaction which was split, and which used a write unlocked, write locked, write partial, or write partial locked transaction, has been completed. The write data was transmitted in the original write transaction that was split.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If any of the modules which received this request have not completed the requested write, they notify the requester by asserting **sr*** during this transaction so that the requester knows that there are still responses outstanding.

6.1.16.12 Read response

A *read response* transaction is the response to a previous request in which a module requested data using one of the following transactions: read un-locked, read partial, write locked with a lock command, or write partial locked with a lock command. The transaction includes a data phase in which data are written to the requesting module. Either the compelled or the packet protocol may be used for this transaction. The requester's global identification and transaction ID is transmitted in the connection phase and status is transmitted in the disconnection phase. There is only one read response returned for a single split read request.

6.1.16.13 Write no acknowledge

The *write no acknowledge* transaction is used by the master to write to one or more slaves. Slaves do not convert the transaction into a split transaction since the master has specified that it does not want a response. Slaves may forward the transactions to other nodes without maintaining a connection with the master.

6.1.16.14 Read invalid

Read invalid may be used by a master, such as a DMA I/O controller, to read a cache line from cache coherent memory. No valid copy is in the master's cache before or after the transaction.

The repository of last resort (generally main memory) snarfs read invalid transactions if intervention is signaled.

6.1.16.15 Write invalid

Write invalid may be used by a master, such as a DMA I/O controller, to write a cache coherent copy of a cache line. No valid copy is in the master's cache before or after the transaction. Modules are not allowed to snarf this transaction.

6.1.16.16 Copyback

The *copyback* transaction is used to write a modified cache line back into cache coherent memory to free up space. The master must have the exclusive modified attribute before issuing the copyback transaction. Invalid caches that want the data may snarf it as it goes to memory. Modules are not allowed to split this transaction.

6.1.16.17 Read shared

Read shared is used by a cached master or bus bridge to service a read miss by reading a cache line from cache coherent memory. A valid read only copy is kept in the master's cache. The copy of the cache line is marked shared unmodified. The copy of the cache line may become valid exclusive unmodified if no other observer reports that it also has a copy.

The repository of last resort (generally main memory) snarfs read shared transactions if intervention is signaled.

6.1.16.18 Read modified

Read modified is used by a caching master or bus bridge to service a write miss by reading a cache line from cache coherent memory. A valid read write copy is kept in the master's cache. The copy of the cache line is

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

marked exclusive modified. All other caches are invalid after the completion of the read modified and any associated modified response.

6.1.16.19 Invalidate

Invalidate is used by a caching master or bus bridge to service a write hit to a valid shared cache line from cache coherent memory by invalidating all other caches and to promote its line from shared to a valid exclusive modified copy. All other caches are invalid after the completion of the invalidate.

While the original master is waiting for a split response, another master on another bus segment may receive write permission. In this case, the modified response transaction to the original master must contain the up-to-date copy of the cache line.

6.1.16.20 Shared response

Shared response is used in response to a split read invalid or read shared. One shared response may satisfy any number of pending or waiting read shared or read invalids.

The repository of last resort (generally main memory) snarfs shared response transactions if there is an intervening cache. Shared response transactions cannot be split.

6.1.16.21 Modified response

Modified response is used in response to a split read modified, invalidate, or write invalid. A bus bridge which is the master of the modified response guarantees that there are no remote valid cached copies of the cache line on the other side of it. Modified response is also used by memory modules to resolve read write races between adjacent buses. Modified response transactions can be split if caches, cache agents, or memory agents have not completed invalidation. Split cached operations are further described in clause 8.

6.1.17 Bus signal descriptions

6.1.17.1 AS* address sync

The master asserts **as*** to indicate that **AD[ ]***, **BP[ ]***, **TG[ ]***, **TP***, **CM[ ]***, and **CP*** have been activated with valid address and command information and that a transaction is being initiated.

The master releases **as*** to indicate that the transaction is being terminated. If the transaction is address only, has a compelled data phase, or uses split response, then the release of **as*** indicates that **CM[ ]*** and **CP*** have been activated with a valid disconnection command. If the transaction uses a split response, then the release of **as*** also indicates that **AD[31..0]*** and **BP[3..0]*** have been activated with a valid requestor's global identification, priority, and status.

6.1.17.2 AK* address acknowledge

The master and the slave(s) assert **ak*** when they detect **AS*** asserted.

Modules release **ak*** after they detect **AS*** released, they have acquired any valid information transmitted by the master, they have activated **te*** with valid error information, and they have asserted **ai***.

6.1.17.3 AI* address acknowledge inverse

The master and the slave(s) assert **ai*** when they detect **AS*** released.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Modules release **ai*** after they detect **AS*** asserted, they have acquired any valid information transmitted by other modules, they have activated **st[7..1]***, **ca[2..0]***, and **di*** in response to the address and command transmitted by the master, and they have asserted **ak***.

6.1.17.4 DS* data sync

The master asserts or releases this signal during the time **as*** is asserted to indicate **CM[ ]*** and **CP*** have been activated with valid command information and the master requests the transfer of another word or packet. During a write transaction the transition of **ds*** also indicates **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP*** have been activated with valid information.

If the master has completed the data transfers for the current transaction and **ds*** is asserted it will first release **as*** and then wait until it detects **DK*** released before releasing **ds***.

6.1.17.5 DK* data acknowledge

During broadcast and intervention transactions in which a slave is the source of the data, the slave asserts **dk*** after it has activated the data on **ad[ ]*** and **d[ ]*** in response to the assertion of **DS***. In other transactions participating slaves assert **dk*** when they detect **DS*** asserted.

Slaves release **dk*** after all the following have been satisfied:

1) They detect the release of **DS***,
2) They have activated **st[7..1]*** with valid status information,
3) They have acquired information transmitted by the master on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP*** if the transaction is a write,
4) They have activated **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** if the transaction is a connected read, and
5) They have asserted **di***.

6.1.17.6 DI* data acknowledge inverse

Participating slaves assert **di*** when they detect the assertion of **AS***. During broadcast and intervention transactions in which a slave is the source of the data, the slave asserts **di*** after it has activated the data on **ad[ ]*** and **d[ ]*** in response to the release of **DS***. In other transactions participating slaves assert **di*** when they detect **DS*** released.

Slaves release **di*** after all the following have been satisfied:

1) They detect the assertion of **DS***,
2) They have activated **st[7..1]*** with valid status information,
3) They have acquired information transmitted by the master on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP*** if the transaction is a write,
4) They have activated **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** if the transaction is a connected read, and
5) They have asserted **dk***.

6.1.17.7 CM[7..0]* command field

The master activates the command field with information that the slaves use to determine the type of transaction in progress or information about the transaction. These lines are uniquely defined in each of the transaction phases. These signals are driven only by the master. Time periods when the signals on these lines are undefined are reserved for use by other standards and therefore should be released. Table 6 shows the command signal definitions for the various phases of a transaction. Only in multiple packet mode do command signals change during the data phase of transactions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table 6—Command field encoding

Phase	CM7*	CM6*	CM5*	CM4*	CM3*	CM2*	CM1*	CM0*
Connection	**AW***	**DW1***	**DW0***	**WR***	**TC3***	**TC2***	**TC1***	**TC0***
Compelled unlocked data phase	**DL2***	**DL1***	**DL0***	**SW***	—	—	—	—
Compelled locked data phase	**DL2***	**DL1***	**DL0***	**SW***	—	**LC2***	**LC1***	**LC0***
Packet data phase	**DL2***	**DL1***	**DL0***	**SW***	—	—	**LT***	**PR***
Disconnection phase	**DL2***	**DL1***	**DL0***	**SW***	—	—	**MR***	**LK***

6.1.17.7.1 CM7* command 7

During the connection phase **CM7*** line is defined as Address Width (**AW***). The master asserts **aw*** to indicate that **AD[63..32]*** are activated with valid address information and that the slave(s) should decode a 64-bit address. Slaves incapable of decoding 64-bit addresses do not respond to transactions in which **AW*** is asserted. The master releases **aw*** to indicate that the slave(s) should decode the 32-bit address activated on **AD[31..0]***.

During the data and disconnection phases, the **CM7*** line is defined as the data transfer length bit 2 (**DL2***). The current master activates **DL[2..0]*** to select one of the seven data transfer lengths or the unrestricted transfer length, as shown in table 7.

This data transfer length refers to the number of transfers the master is requesting, not the number of bytes. An unrestricted data transfer length is not allowed for packet transfers or for split read transactions. The unrestricted data transfer length may be used for split write transactions. If the master specifies a numerical transfer length with a starting address that is not an integer multiple of an address computed by multiplying the data width by the data length, then the master will end the transfer before it crosses an address that is an integer multiple. For example, a transfer with a data width of 32 bits and a transfer length of 8 that starts at address 8_{16} would conclude at address $1C_{16}$. A transfer with a data width of 64 bits and a transfer length of 8 that starts at 10_{16} Would conclude at address 38_{16}.

During compelled transactions the master controls the number of transfers by the number of transitions of **DS***. During compelled transactions slaves may terminate the transaction by asserting **ED*** regardless of the transfer length. When the master specifies a transfer length other than restricted during a compelled transaction, it will never exceed that number of transfers. In general, modules which are not capable of split transactions or packet mode should but are not required to activate **DL[2..0]***

6.1.17.7.2 CM6* command 6

During the connection phase this line is defined as Data Width 1 (**DW1***). The master asserts **dw1*** to indicate that the bus width for the transaction is either 128 bits or 256 bits wide. The master releases **dw1*** to indicate that the transfer is either 32 bits or 64 bits.

During the data and disconnection phases, the **CM6*** line is defined as the data transfer length bit 1 (**DL1***).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7—Data transfer length

DL2*	DL1*	DL0*	Data transfer length
0	0	0	Unrestricted
0	0	1	1
0	1	0	2
0	1	1	4
1	0	0	8
1	0	1	16
1	1	0	32
1	1	1	64

6.1.17.7.3 CM5* command 5

During the connection phase this line is defined as Data Width 0 (**DW0***). The master asserts **dw0*** to indicate that the bus width for the transaction is either 64 bits or 256 bits wide as shown in table 8. The master releases **dw0*** to indicate that the transfer is either 32 bits or 128 bits.

During the data and disconnection phases, the **CM5*** line is defined as the data transfer length bit 0 (**DL0***).

Table 8—Data path width

DW1*	DW0*	Data path width
0	0	32 bits
0	1	64 bits
1	0	128 bits
1	1	256 bits

6.1.17.7.4 CM4* command 4

During the connection phase this line is defined as Write (**WR***). The master asserts **wr*** to indicate that data is to be transferred from the master to the slave(s) during the current transaction. The master releases **wr*** to indicate that data is to be transferred to the master from the slave(s).

During the data and disconnection phases this line is defined as Slave Write (**SW***). The master asserts **sw*** when it detects the **IV*** status line asserted at the end of the connection phase or after each slave packet handshake during packet mode. The assertion of **SW*** indicates to the selected slave (shared memory) that it is to receive the data during the current transfer or packet instead of sourcing the data.

6.1.17.7.5 CM3* command 3

During the connection phase this line is defined as Transaction Command 3 (**TC3***). The master activates **tc[3..0]*** and **wr*** to select one of the transaction commands in table 9.

During the data and disconnection phases this line is not defined and is to be released.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table 9—Transaction command encoding

Transaction	TC3*	TC2*	TC1*	TC0*	WR*
Read unlocked	0	0	0	0	0
Write unlocked	0	0	0	0	1
Address only unlocked	0	0	0	0	1
Read locked	0	0	0	1	0
Write locked	0	0	0	1	1
Address only locked	0	0	0	1	1
Read partial	0	0	1	0	0
Write partial	0	0	1	0	1
Read partial locked	0	0	1	1	0
Write partial locked	0	0	1	1	1
-reserved-	0	1	0	0	0
Write response	0	1	0	0	1
-reserved-	0	1	0	1	0
Read response	0	1	0	1	1
-reserved-	0	1	1	0	0
Write no acknowledge	0	1	1	0	1
-reserved-	0	1	1	1	0
-reserved-	0	1	1	1	1
Read invalid	1	0	0	0	0
Write invalid	1	0	0	0	1
Read shared	1	0	0	1	0
Copyback	1	0	0	1	1
-reserved-	1	0	1	0	0
-reserved-	1	0	1	0	1
Read modified	1	0	1	1	0
Invalidate	1	0	1	1	1
-reserved-	1	1	0	0	0
Shared response	1	1	0	0	1
-reserved-	1	1	0	1	0
-reserved-	1	1	0	1	1
-reserved-	1	1	1	0	0
-reserved-	1	1	1	0	1
-reserved-	1	1	1	1	0
Modified response	1	1	1	1	1

6.1.17.7.6 CM2* command 2

During the connection phase this line is defined as Transaction Command 2 (**TC2***).

During the data phase of a locked transaction this line is defined as Lock Command 2 (**LC2***). The master activates **lc[2..0]*** to select one of seven possible lock commands as shown in table 10.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

During the disconnection phase this line is not defined and is to be released.

Table 10—Locked command encoding

LC2*	LC1*	LC0*	Lock command
0	0	0	- No command -
0	0	1	Mask & swap
0	1	0	Compare & swap
0	1	1	Fetch & add big
1	0	0	Fetch & add little
1	0	1	- reserved -
1	1	0	- reserved -
1	1	1	- reserved -

6.1.17.7.7 CM1* command 1

During the connection phase this line is defined as Transaction Command 1 (**TC1***).

During the data phase of a locked transaction, this line is defined as Lock Command 1 (**LC1***).

During the data phase of a packet transfer, this line is defined as Last Transaction flag (**LT***). The master asserts **lt*** when it detects the **TF*** status line asserted at the end of the connection phase or after each slave packet handshake.

During the disconnection phase this line is defined as More (**MR***). The master asserts More to indicate that a block copy operation is in effect and pre-fetching, write reordering, and write buffering are permitted. See IEEE Std 896.3-1993 for more information on this facility.

6.1.17.7.8 CM0* command 0

During the connection phase this line is defined as Transaction Command 0 (**TC0***).

During the data phase, during a locked transaction, this line is defined as Lock Command 0 (**LC0***).

During the disconnection phase of locked transactions this line is defined as Lock (**LK***). The master asserts **lk*** to indicate that any modules which have locked resources are to maintain the lock past the end of the current transaction.

During the data phase when packet mode is selected this line is defined as Packet Request (**PR***). The master asserts packet request to indicate to the slave(s) that the master is requesting that an additional packet be transferred. During the data phase for unlocked transactions when the packet mode is not selected this line is not defined and is to be released.

6.1.17.8 CP* command parity

A line that carries the parity bit for the **CM[7..0]*** lines, such that if the number of asserted signals in **CM[7..0]*** is even **CP*** is asserted.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.17.9 ST[7..0]* status

One or more status lines may be asserted by one or more participating modules to indicate the slave status and the interpretation of the information received from the master. The following table shows the status signal definitions for the various phases of a transaction. If asserted simultaneously in the connection phase the precedence the master should use to interpret status or capability signals is: **ST6* (BE*)**, **ST2* (SL*)**, **ST7* (WT*)**, **ST1* (BS*)**, **CA2* (SR*)**, **ST4* (TF*)**, **ST5* (IV*)**, **ST3* (BC*)**, **CA1* (CO*)**, **CA0* (TS*)**.

Table 11—Status encoding

Phase	ST7*	ST6*	ST5*	ST4*	ST3*	ST2*	ST1*	ST0*
Connection	WT*	BE*	IV*	TF*	BC*	SL*	BS*	TE*
Data	WT*	BE*	IV*	TF*	BC*	SL*	ED*	TE*
Disconnection	WT*	BE*	IV*	TF*	BC*	SL*	BS*/ ED*	TE*

6.1.17.9.1 ST7* status 7

This line is defined as Wait (**WT***).

A selected or participating slave asserts **wt*** during the connection phase to end the transaction without a data phase. The master then waits for a system-defined event before retrying the transaction. No change in cache attributes is to occur if **WT*** is asserted during a transaction. Only the master is able to detect the state of **WT*** during the connection phase. The slave continues to assert **WT*** through the disconnection phase. All modules are able to detect the state of **WT*** during the disconnection phase.

6.1.17.9.2 ST6* status 6

ST6* is defined as Beat Error (**BE***).

During the connection phase, any module asserts **be*** if it detects a parity error on **CM[ ]***, **AD[ ]***, or **TG[ ]***. A participating module asserts **be*** during the connection phase if it detects an illegal operation. Modules with 64-bit address capability also check parity on **AD[63..32]*** if **AW*** is asserted. Modules also assert **be*** if the master requires them to respond with a capability which they lack. If the master detects **BE*** asserted during the connection phase, then it proceeds to the disconnection phase.

During the compelled data phase any participating slave asserts **be*** if it detects a parity error on **CM[ ]***, **AD[ ]***, **D[ ]***, or **TG[ ]***. Modules also check parity on **AD[63..32]*** and **D[ ]***, if the appropriate data width (**DW[ ]***) command bits were asserted. Due to pipelining **BE*** is not guaranteed to be asserted in the data beat that the error was detected.

The master can continue handshakes following the assertion of **BE***, but the data is ignored by both the master and slaves.

During the disconnection phase, modules that asserted **be*** in the connection or data phase continue to assert **be***.

6.1.17.9.3 ST5* status 5

This line is defined as Intervention (**IV***).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

During compelled transactions a module asserts **iv***, if it is intervening in the transaction. The module maintains the assertion of **iv*** until the end of the disconnection phase. When the master detects **IV*** asserted it enables its wire-OR integrators on its receivers of **DK*** and **DI*** and keeps them enabled for the duration of the transaction.

During a packet transaction a module asserts **iv*** if it is intervening in the packet currently being requested. The module maintains the assertion of **iv*** until the next transition of **DS*** or until the end of the disconnection phase.

6.1.17.9.4 ST4* status 4

This line is defined as Transaction Flag (**TF***).

Higher level protocols, such as cache coherence and message passing, define the information carried by this line. The definition of the signal can vary on a transaction by transaction basis.

6.1.17.9.5 ST3* status 3

This line is defined as Broadcast (**BC***).

Modules assert **bc*** during the connection phase to convert a single slave transaction into a multiple slave transaction. When the master detects **BC*** asserted it enables its wire-OR integrators on its receivers of **DK*** and **DI*** and keeps them enabled for the duration of the transaction. Modules maintain the assertion of **bc*** until the end of the disconnection phase.

6.1.17.9.6 ST2* status 2

This line is defined as Selected (**SL***).

Modules assert **sl*** during the connection phase when they recognize the address as one they should respond to and they do not detect any parity errors. If the master does not detect **SL*** asserted at the end of the connection phase it proceeds directly to the disconnection phase since none of the bus modules have responded.

6.1.17.9.7 ST1* status 1

During the connection phase, **ST1*** is defined as Busy (**BS***).

If the line was asserted during the connection phase it is also defined as **BS*** during the disconnection phase. A selected or participating slave asserts **bs*** to end the transaction without a data phase. The master then waits for a system-defined period of time before retrying the transaction. No change in cache attributes is to occur if **BS*** is asserted during a transaction. Only the master is able to detect the state of **BS*** during the connection phase. All modules are able to detect the state of **BS*** during the disconnection phase.

If **BS*** was not asserted during the connection phase, **ST1*** is defined as End of Data (**ED***) during the data and disconnection phases. A participating module asserts **ed*** during a compelled transaction to indicate that the current transaction should be terminated without further data transfers and the master is to proceed to the disconnection phase. A participating module asserts **ed*** during a multiple packet mode transaction to indicate that the current transaction must be terminated after the packet corresponding to the last one acknowledged without **ED*** asserted is transferred. In most cases the master will initiate a new transaction after receiving an asserted **ED*** if it has more data to transfer. If the master was performing a read response transaction and received an asserted **ED***, then it will discard any data remaining in the response and not initiate a new transaction. During the disconnection phase **ED*** indicates end of data status for the last compelled data transfer or the last requested packet.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The master can continue handshakes following the assertion of **ED***, but the data is ignored by both the master and slaves.

6.1.17.9.8 ST0* status 0

This line is defined as Transaction Error (**TE***).

TE* is not valid during the connection phase. During the data or disconnection phase, modules assert **te*** to report errors such as parity, packet overrun, packet underrun, the master requesting packets beyond an end of data status, and disconnection parity. Modules maintain the assertion of **te*** until the end of the connection phase of the next transaction, independent of change of tenure. A module that asserted beat error (**be***) during the connection or data phase also asserts **te*** during the disconnection phase. Although **TE*** can be asserted during the data phase it can not be sampled until the end of the disconnection phase when **AKf** is released.

6.1.17.10 CA[2..0]* capability

One or more capability lines are asserted by one or more modules to indicate the capabilities of the participating modules. Table 12 shows the capability signal definitions for the various phases of a transaction.

Table 12—Capability encoding

Phase	CA2*	CA1*	CA0*
Connection	**SR***	**CO***	**TS***
Data	**SR***	**CO***	**TS***
Disconnection	**SR***	**CO***	**TS***

6.1.17.10.1 CA2* capability 2

This line is defined as Split Response (**SR***).

A module asserts **sr*** to require that the current transaction be split into multiple transactions or to indicate during a response that there are still responses outstanding for the data requested.

6.1.17.10.2 CA1* capability 1

This line is defined as Compelled (**CO***).

A module asserts **co*** to require that the current transaction use the compelled protocols instead of the packet protocols.

6.1.17.10.3 CA0* capability 0

This line is defined as Transmit Speed (**TS***).

A module asserts **ts*** to request that the lower of the two software-selected packet-transmit speeds be used during the current transaction. If **co*** is asserted then **ts*** should be released.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.17.11 AD[31..0]* address/data/lane deselect/return address

During the connection phase of most transactions, **AD[31..5]*** are activated with the address for the current transaction and **AD[1..0]*** are user-defined. **AD[4..2]*** contain either address or user-defined information depending upon the data width (**DW[ ]***) selected.

The only exception is during the connection phase of read response and write response transactions. In this case:

1) **AD[31..28]*** are set to one.
2) **AD[27..12]*** are activated with the requestor's global identification.
3) **AD[11..6]*** are reserved.
4) **AD[5..0]*** are activated with a transaction identification which is user defined.

During the data phase of write transactions, these lines (**AD[31..0]***) are activated with write data by the master. During the data phase of read transactions, these lines are activated with read data by one or more slaves.

Data accesses of 32 bits are always transferred on **AD[31..0]*** independent of the actual width of a module.

Bytes are transferred in their natural positions with no justification. Byte ordering is specified in related Futurebus+ profiles.

During the first odd beat of partial transfers, **AD[31..0]*** are defined as lane deselect signals (**L[ ]***) for the transfer occurring during the subsequent beats. A partial transfer consists of one beat of lane disables and one or more beats of data. During partial write transfers, if a lane deselect signal is asserted, the corresponding byte lane is not modified by the receiving slave. During partial read transfers, if a lane deselect signal is asserted by the master, the corresponding byte lane is not evaluated by the master. Lane deselect signals corresponding to lanes not enabled by the data width command may be released. Byte parity is supplied and checked across the entire **AD[ ]*** and **D[ ]*** fields enabled by the data width command during the lane disable beat.

During the disconnection phase of connected read transactions, **AD[31..0]*** may still be driven with data from the last data beat. Therefore, disconnection information can not be placed on these lines and parity is not checked.

During the disconnection phase of read response and write response transactions:

1) **AD[31..16]*** are reserved.
2) **AD[15..8]*** are activated with the requestor's original transaction priority. The arbitration priority should not be used because it may be inherited.
3) **AD[7..3]*** are reserved.
4) **AD2*** contains the selected status **SL***. This bit is asserted by the module at the request's ultimate destination, as signaled in the request's address.
5) **AD1*** contains the conflict status. This bit is asserted by the module at the request's ultimate destination when this module detects a potential deadlock condition that requires the request transaction to be backed off all the way back to the request's originator. This is an end-to-end busy, and the requester should not expect further responses.
6) **AD0*** contains the transaction error status **TE***. This bit is asserted if an error condition was detected anywhere along the request transaction's path. The requester should not expect further responses.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

During the disconnection phase of shared response and modified response transactions:

1) **AD[31..28]*** are set to one.
2) **AD[27..12]*** are activated with the requestor's global identification.
3) **AD[11..4]*** are activated with the requestor's original transaction priority.The arbitration priority should not be used because it may be inherited.
4) **AD3*** is reserved.
5) **AD2*** contains the selected status **SL***. This bit is asserted by the module at the request's ultimate destination, as signaled the request's address.
6) **AD1*** contains the conflict status. This bit is asserted by the module at the request's ultimate destination when this module detects a deadlock condition that requires the request transaction to be backed off all the way back to the request's originator. This is an end-to-end busy and the requester should not expect further responses.
7) **AD0*** contains the transaction error status **TE***. This bit is asserted if an error condition was detected anywhere along the request transaction's path. The requester should not expect further responses.

During the disconnection phase of those transactions not described above:

1) **AD[31..16]*** are activated with the requestor's global identification.
2) **AD[15..8]*** are activated with the requestor's original transaction priority. The arbitration priority should not be used because it may be inherited.
3) **AD[7..6]*** are reserved.
4) **AD[5..0]*** are activated with a transaction identification.

6.1.17.12 AD[63..32]* address/data

During the connection phase of transactions with **AW*** asserted **AD[63..32]*** are activated with the most significant 32 bits of address for the current transaction.

During the data phase of write transactions with either **DW1*** or **DW0*** asserted, these lines are activated with write data by the master. During the data phase of read transactions with either **DW1*** or **DW0*** asserted, these lines are activated with read data by one or more slaves.

During the disconnection phase, these lines are not defined. Additional capabilities of these lines during the disconnection phase may be defined in another standard.

6.1.17.13 D[255..64]* data

During the connection phase **D[255..64]*** are not defined.

During the data phase of write transactions with **DW1*** asserted, **D[127..64]*** are activated with the write data by the master. During the data phase of write transactions with **DW1*** and **DW0*** asserted, **D[255..128]*** are activated with the write data by the master. During the data phase of read transactions with **DW1*** asserted, **D[127..64]*** are activated with the read data by the slave. During the data phase of read transactions with **DW1*** and **DW0*** asserted, **D[255..128]*** are activated with read data by one or more slaves.

During the disconnection phase, these lines are not defined. Additional capabilities of these lines during the connection and disconnection phases may be defined in another standard.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.17.14 BP[31..0]* byte parity

Each byte parity line carries the parity bit for one of the byte lanes, such that if the number of asserted signals in each byte is even, the byte parity signal is asserted.

BP0* carries the parity for **AD[7..0]***, **BP1*** corresponds to **AD[15..8]***, and so on through **BP31*** corresponding to **D[255..248]***.

6.1.17.15 TG[7..0]* tag

The **TG[7..0]*** lines carry optional additional information related to that on the **AD[63..0]*** and **D[255..64]***. The information represented by the signals on these lines is not constrained by this International Standard. Recommendations on how they might be used may be provided in a future standard. The number of **TG[ ]*** lines active is negotiated during initialization and is contained in the Module Logical Capability register as described in 7.1.

6.1.17.16 TP* tag parity

The tag parity line carries the parity bit for **TG[7..0]*** such that if the number of asserted signals is even **TP*** is asserted.

If no tag lines are active (based on the control and status register selection), this line is released.

6.1.17.17 ET* end of tenure

The current master releases **et*** to indicate that its bus tenure has ended. The release of **ET*** is the indication that the current master has released all of the parallel bus lines and that the master elect may begin a bus transaction.

The master asserts **et*** during its first bus transaction when it detects **AIf** at logic zero and prior to the release of **as***.

The release of **ET*** also makes a change in tenure visible to all modules causing them to release any resource locks.

6.1.18 Bus beats

A *beat* consists of the transition on a synchronization line made by a master, followed by the release of an acknowledge line by one or more slaves. Command and data information may be sent from the master to the slave(s) during the first half of a beat, and status, capability, and data information may be returned to the master from the slave(s) during the second half. This type of handshake is referred to as compelled, because the slave is compelled to provide a response before the master will proceed to the next beat. Transactions on the bus consist of a collection of these beats carried out by the master and one or more slaves.

6.1.18.1 Single slave bus beat

Figure 17 shows a general beat in which information is transferred between the master and a single slave. A detailed description of the operation follows:

1) The master first activates the information lines with the appropriate information signals.
2) The master then waits for the skew time of its own transmitters and the bus signaling environment to ensure that all its information signals are valid on the bus prior to the assertion of its synchronization signal on the bus.
3) The master then asserts its synchronization signal.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4) The slave then detects the assertion of the master synchronization signal.
5) The slave then waits for the skew of its own receivers.
6) The slave then captures the information from the master. (This capture includes whatever additional time may be necessary to account for setup and hold times within its data capture circuits.)

NOTE—This method of synchronization ensures the technology independence of each module, because each need only account for its own skews and timing requirements.

7) The slave then activates the information lines with the appropriate response information signals.
8) The slave then waits for the skew time of its own transmitters and the bus signaling environment to ensure that all its information signals are valid on the bus prior to the release of its synchronization signal on the bus.
9) The slave then releases its synchronization signal.
10) The master then detects the release of the slave synchronization signal.
11) The master may then activate the information lines with new information. The master then waits for the skew of its own receivers.
12) The master then captures the response information from the slave. (This capture includes whatever additional time may be necessary to account for setup and hold times within its data capture circuits).

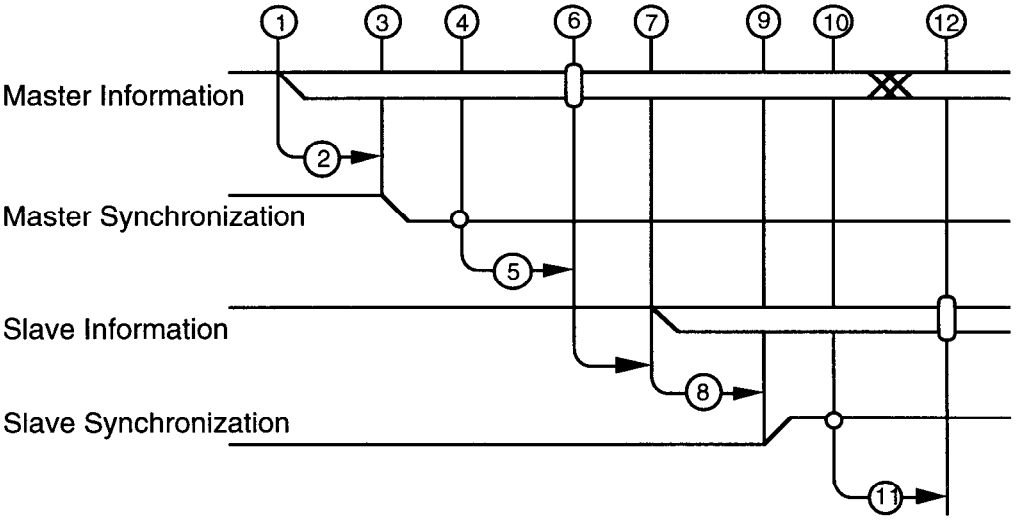


Figure 17—Single slave bus beat

6.1.18.2 Multiple slave bus beat

Figure 18 shows a general beat in which information is transferred between a master and multiple slaves. When multiple slaves are involved, a transfer is known as a broadcast transfer. The difference between a multi-slave transfer and a single slave transfer involves the slave synchronization signal. Initially, all of the slaves involved in the beat must be asserting the synchronization signal. Steps 1)–6) are identical to those of 6.1.18.1. Steps 7)–10) are as follows:

7) Slaves place their information on the bus.
8) Slaves wait for their skew.
9) Each slave releases its synchronization signal.
10) Since the slave synchronization signal that appears on the bus is the wire-OR of all slaves asserting this signal, all slaves must release this signal before it appears released on the bus. This allows the slowest of the slaves to control the time of the release. A fundamental transmission line effect called

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

the wire-OR glitch is caused by this situation. The recipient of a signal used in this way must act upon it only after it has passed through a wire-OR integrator. This causes multiple slave beats to be inherently slower than single slave beats.

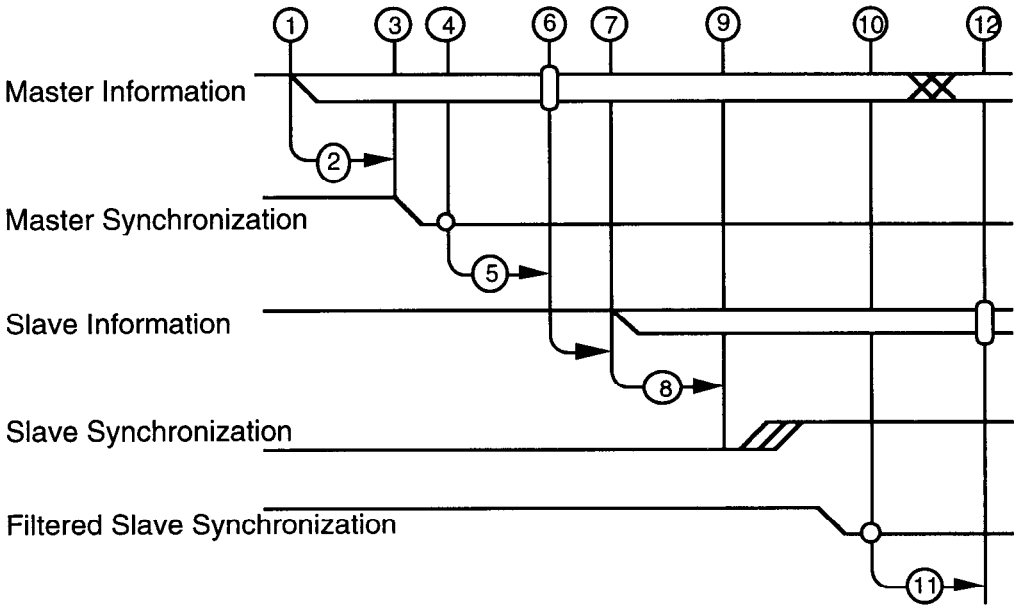


Figure 18—Multiple slave bus beat

6.1.19 Transaction examples

6.1.19.1 Address only transaction

An address only transaction consists of two consecutive beats. The first beat serves to establish a connection between the master and one or more slaves and is therefore referred to as the connection phase of the transfer. The second beat serves to break a connection between the master and the connected slaves and is referred to as the disconnection phase. Figure 19 shows an address only transaction.

Bus beats in which the master asserts its synchronization line are referred to as *odd beats*. Conversely, bus beats in which the master releases its synchronization line are referred to as *even beats*. Protocols in which both edges of a synchronization signal are used are referred to as *two edge handshakes*.

This transfer is a basic bus primitive which provides the master with two information time slots in which it may transmit information to one or more slaves. Two information time slots are also provided for the master to receive information from the slave(s) in response.

The master first activates the **AD[ ]*** lines with the address, the **CA [ ]*** lines with the appropriate capability information, the **CM[ ]*** lines with the appropriate connection command bits, the **TG[ ]*** lines with the appropriate tag information, and the appropriate parity lines, and then asserts **as*** to mark their validity. When the slaves detect **AS*** asserted, they assert **ak*** and evaluate the address and the command information. If the address is one which they should respond to as a selected slave or a participating slave, they assert **di***. The assertion of **ak*** and **di*** in the connection phase is simply to prepare them for later use. Selected slaves assert **sl*** as shown to notify the master that they are responding to the transaction. The slaves also assert any of the other status and capability lines that are applicable. The slaves then release **ai***. Since not all of the slaves release **ai*** at the same instance, a wire-OR glitch can occur. The master therefore

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

observes **AI*** through a wire-OR integrator. When the master detects **AIf** at a logic zero it evaluates the status and capability lines.

NOTE—The master may have chosen to do an address only transaction in advance or may find it necessary to revert to an address only transaction based on the status and capability information. For example, if the master detected **SR*** asserted it would convert a read transaction to a read request.

In the case of a split response transaction, the master activates **AD[ ]*** and **BP[ ]*** with disconnection data, **CM[ ]*** and **CP*** with the disconnection command, and **TG[ ]*** and **TP*** with the appropriate tag information. The master then releases **as*** to mark the validity of **AD[ ]***, **BP[ ]***, **TG[ ]***, **TP***, **CM[ ]***, and **CP*** and to indicate to the participating slaves that the transaction is being terminated. The slaves, upon detecting the release of **AS*,** evaluate the information on **AD[ ]***, **BP[ ]***, **TG[ ]***, **TP***, **CM[ ]***, **CP***, and **ST[ ]*** and then assert **ai*** and release **ak*** and **di***. When the master and slaves detect the **AKf** at logic zero, they release **AD[ ]***, **BP[ ]***, **TG[ ]*, TP***, **CM[ ]***, **CP***, **CA[ ]***, and **ST[ ]*** and the transaction ends.

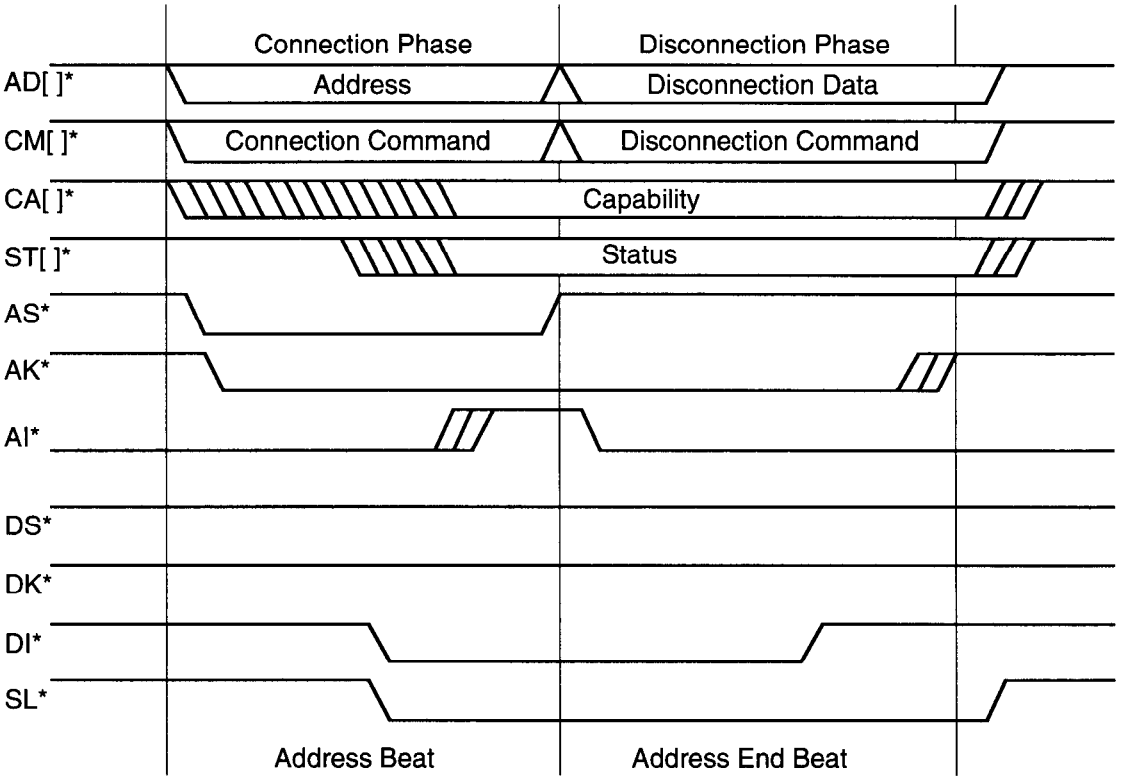


Figure 19—Address only transaction

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.2 Compelled read

The compelled read transaction (see figure 20) begins with a connection phase as in the address only transaction. After the master detects **AIf** at a logic zero and evaluates **CA[ ]*** and **ST[ ]***, it activates **CM[ ]*** with the appropriate data phase command and releases **AD[ ]*** and **TG[ ]**. The master then asserts **ds*** to request data from the slave. When the selected slave detects **DS*** asserted, it evaluates **CM[ ]*** and asserts **dk***. When the requested data is available, the slave activates **AD[ ]*** and **TG[ ]***. The slave then releases **di*** to notify the master that the data is valid. When the master detects **DI*** released it captures the data and releases **ds*** to request data from the next address. When the slave detects the release of **DS***, it asserts **di***. When the requested data is available, the slave activates **AD[ ]*** and **TG[ ]*** and, for example, asserts **ed*** to indicate that it can support no further data transfers past the current one. The slave then releases **dk***. When the master detects the release of **DK***, it captures the data on **AD[ ]*** and proceeds to the disconnection phase as in the address only transaction.

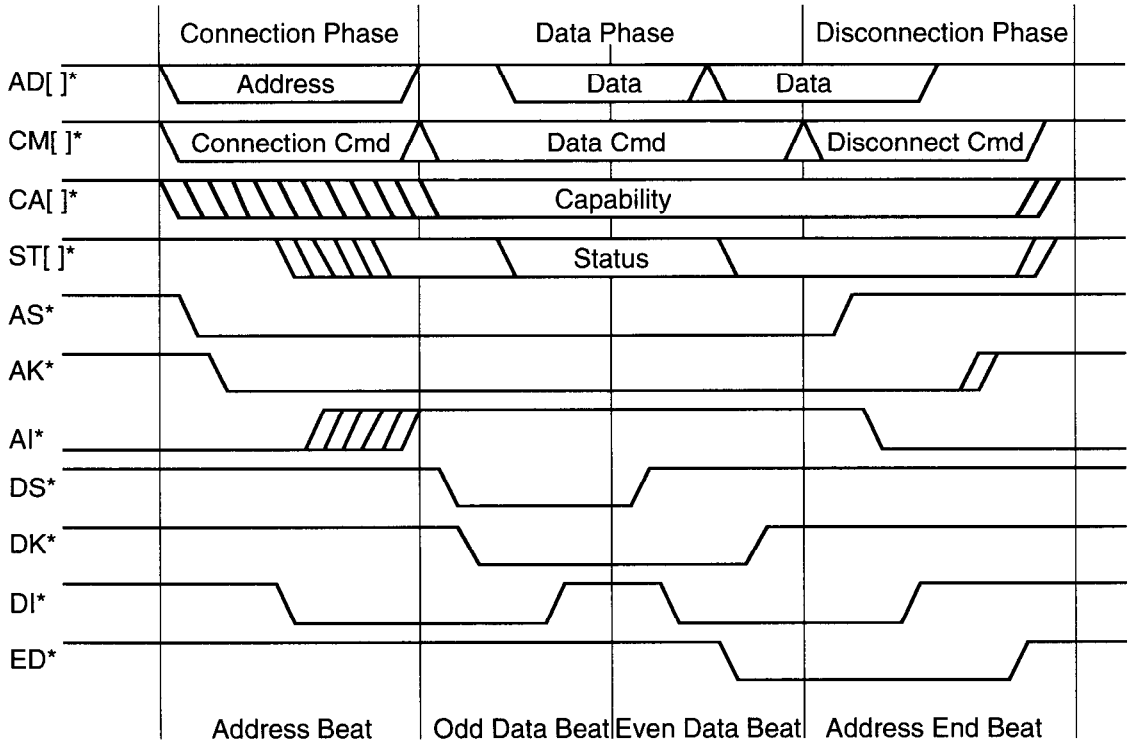


Figure 20—Compelled read transaction

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.3 Compelled read broadcast/broadcall

The compelled read transaction with broadcast differs from the previous transaction in that one or more of the slaves assert **bc*** during the connection phase. This indicates to the master that it should observe **DI*** and **DK*** after they have passed through wire-OR integrators due to the fact that multiple slaves are releasing the signals. In this transaction the master requests a single transfer. This leaves both **DS*** and **DK*** asserted upon entry to the disconnection phase. In this situation the slaves release **dk*** when they detect the release of **AS***. The master releases **DS*** when it detects **DKf** at logic zero. This restores **DS*** to its released state.

This same protocol is used for two possible transactions: *broadcall* and *broadcast.*

The broadcall version (see figure 21) of the transaction occurs when multiple slaves respond with data and the master receives the wire-OR of data from all of the slaves. For example, if each module only asserted one of the bus lines, status could be received from all modules at once.

The broadcast version of the transaction occurs when one of the slaves sources the data and one or more other slaves receive the data along with the master. In the read broadcast transaction, only the module sourcing the data initially asserts **dk*** which indicates the data is valid. The other modules wait until they detect **DK*** asserted before asserting **dk***.

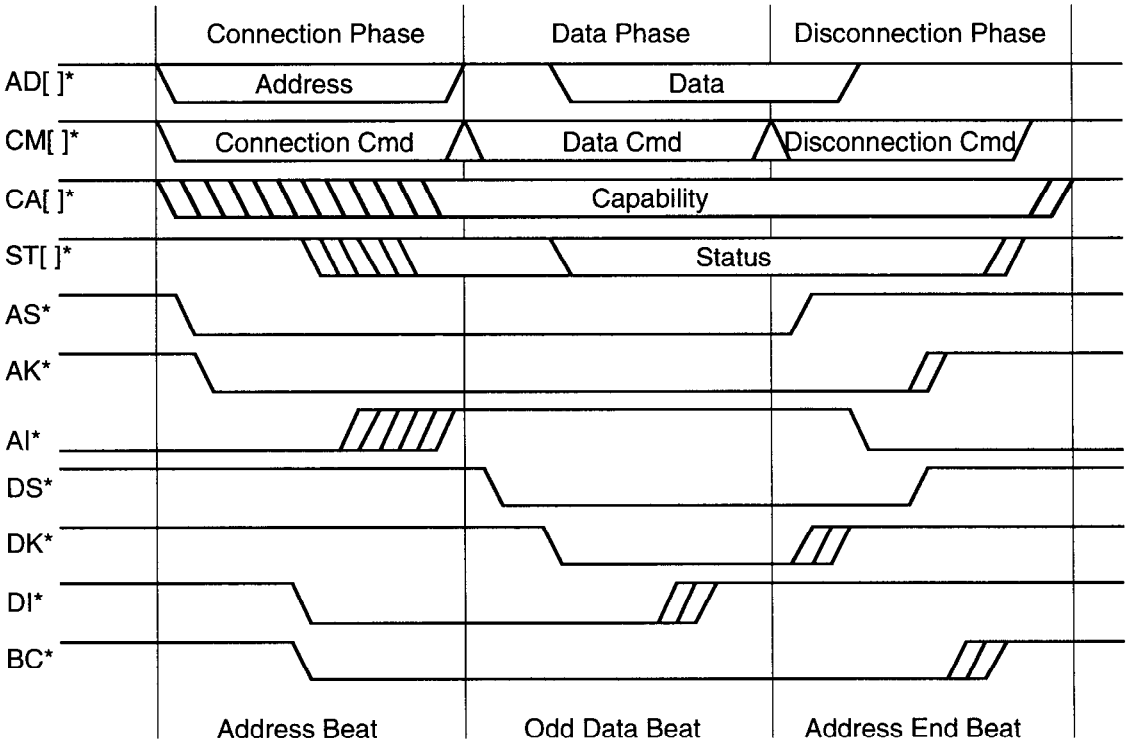


Figure 21—Compelled read transaction with broadcall

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.4 Compelled read intervention

In a compelled read transaction with intervention (see figure 22), a slave asserts **iv*** during the connection phase to indicate that it has a more up-to-date copy of the data requested by the master, than the selected slave does. The master upon detecting **IV*** asserted responds by asserting **sw*** during the data phase. The selected slave upon detecting **SW*** asserted prepares to receive the data instead of sourcing it. Since more than one slave is involved the master must observe **DK*** and **DI*** after they pass through wire-OR integrators.

In the read transaction with intervention, only the module sourcing the data initially asserts **dk***. The other modules wait until they detect **DK*** asserted before asserting **dk***. The same is true in even data beats except that the module sourcing the data initially asserts **di***. The other modules wait until they detect **DI*** asserted before asserting **di***.

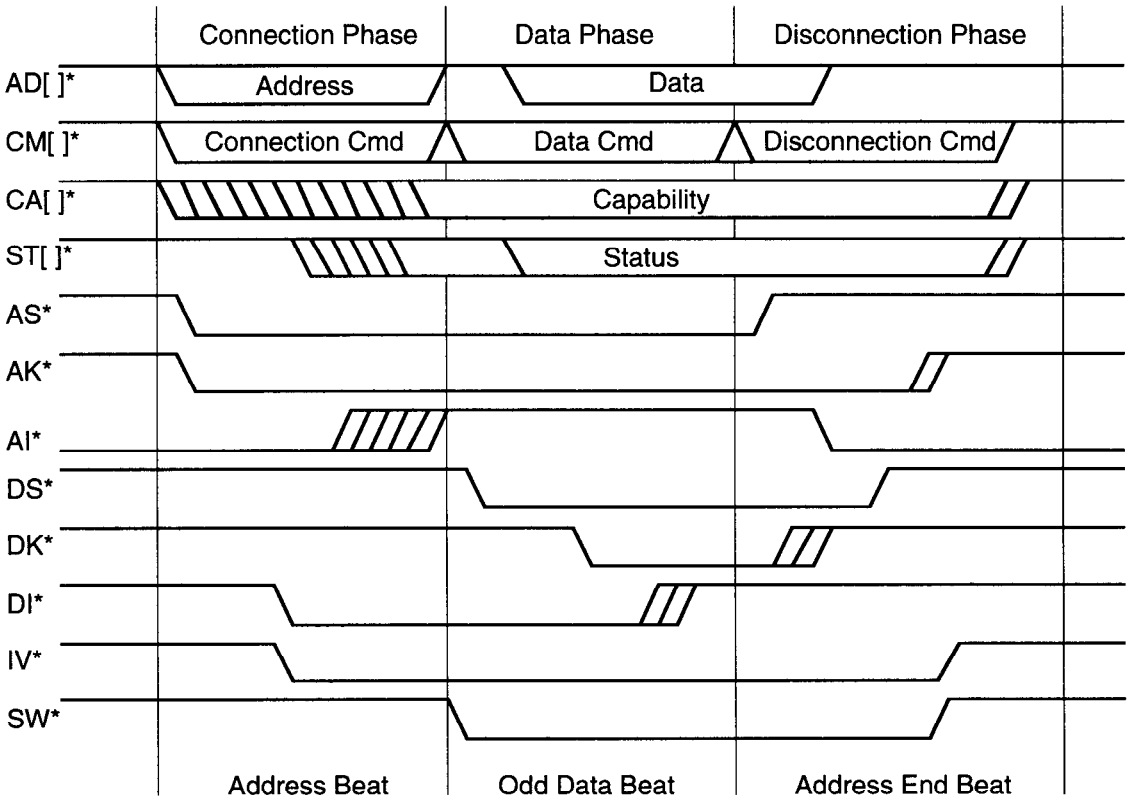


Figure 22—Compelled read transaction with intervention

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.19.5 Compelled read partial

The compelled read partial transaction (see figure 23) is used to transfer only particular bytes of the data field. In the first odd data beat, the master transfers lane deselect information on **AD[31..0]***. In the even data beat and all subsequent data beats, the slave(s) source the requested data. The master, upon receiving the data, ignores byte lanes if the corresponding lane deselect signal is asserted (see 6.1.17.11).

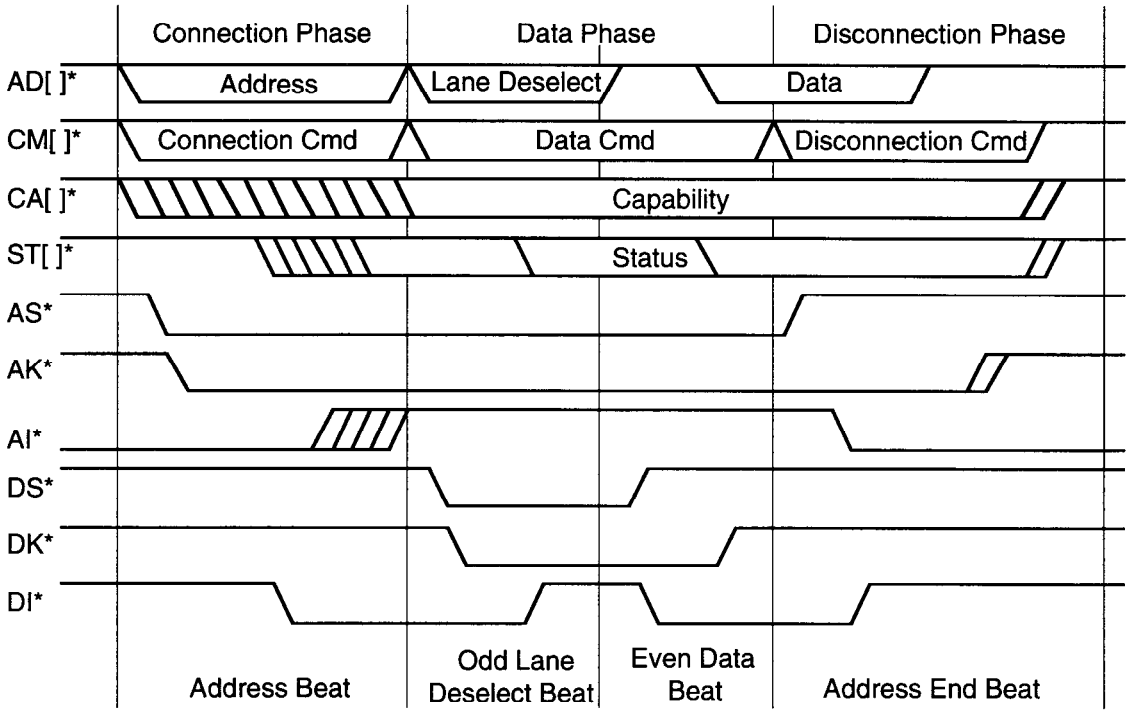


Figure 23—Compelled read partial transaction

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.6 Compelled write

The compelled write transaction (see figure 24) begins with a connection phase as in the address only transaction. After the master detects **AIf** at a logic zero and evaluates **CA[ ]*** and **ST[ ]***, it activates **CM[ ]*** with the appropriate data phase command, **AD[ ]*,** and **D[ ]*** with the write data, and **TG[ ]*** with the tag information. The master then asserts **ds*** to notify the slave that the **CM[ ]***, **AD[ ]***, and **TG[ ]*** are valid. When the selected slave detects **DS*** asserted, it asserts **dk*** and captures the data on **AD[ ]*** and **TG[ ]***. The slave then releases **di*** to notify the master that it has captured the data and that **ST[ ]*** are valid. When the master detects **DI*** released, it captures the slave status, activates **AD[ ]*** with the next write data and **TG[ ]*** with the next appropriate tag information, and releases **ds***. When the selected slave detects **DS*** released, it asserts **di*** and captures the data on **AD[ ]*** and **TG[ ]***. The slave then releases **dk*** to notify the master that it has captured the data and that **ST[ ]*** are valid. When the master detects **DK*** released, it captures the slave status and proceeds to the disconnection phase, if **ED*** was asserted by the slave or all the data has been transferred.

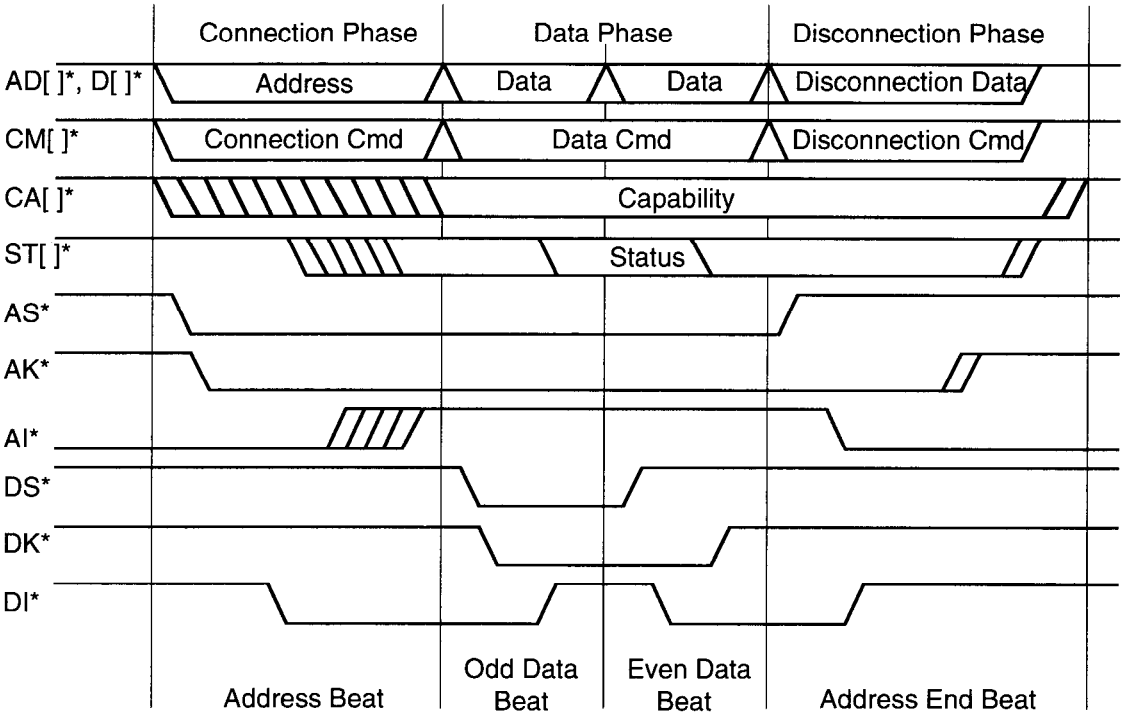


Figure 24—Compelled write transaction

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.19.7 Compelled write broadcast

The compelled write transaction with broadcast (see figure 25) is identical to the previous transaction except that one or more of the slaves assert **bc*** during the connection phase. This indicates to the master that it should observe **DI*** and **DK*** after they have passed through wire-OR integrators due to the fact that multiple slaves are releasing the signals.

All of the participating slaves receive the data from the master in this transaction.

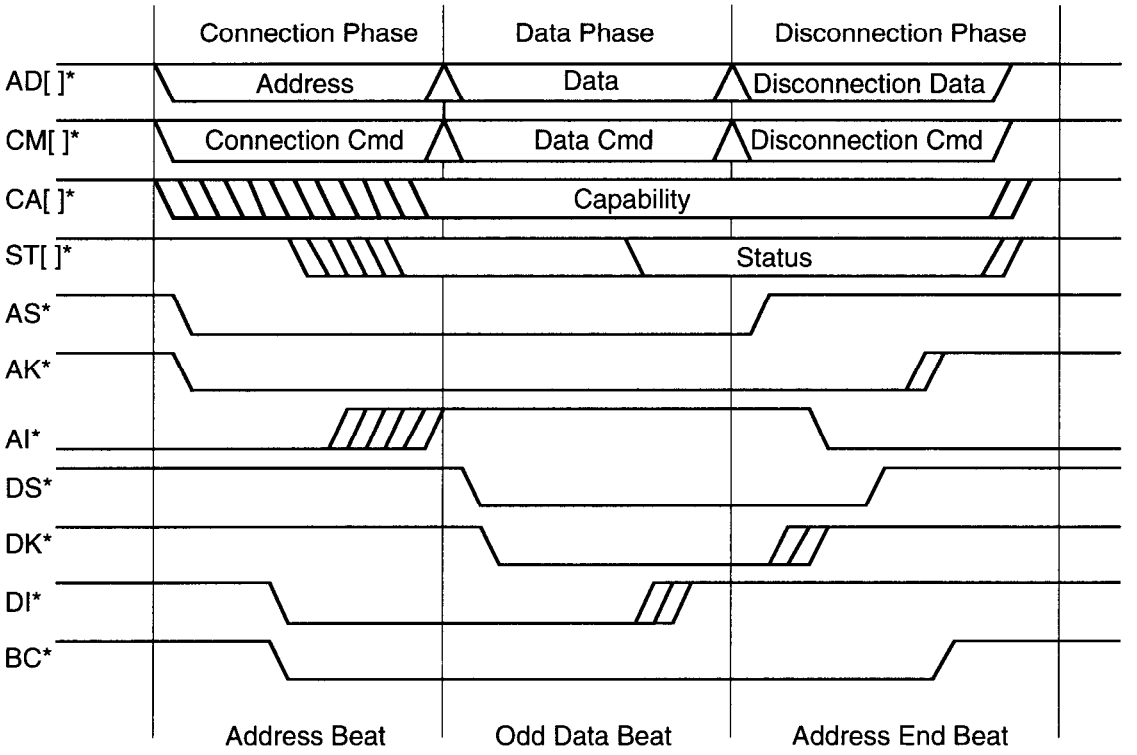


Figure 25—Compelled write transaction with broadcast

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.8 Compelled write partial

The compelled write partial transaction (see figure 26) is used to transfer only particular bytes of the data field. In the first odd data beat, the master transfers lane deselect information on **AD[31..0]***. In the first even and subsequent data beats, the master transfers write data. The slave(s), upon receiving the data, ignores byte lanes if the corresponding lane deselect signal is asserted (see 6.1.17.11).

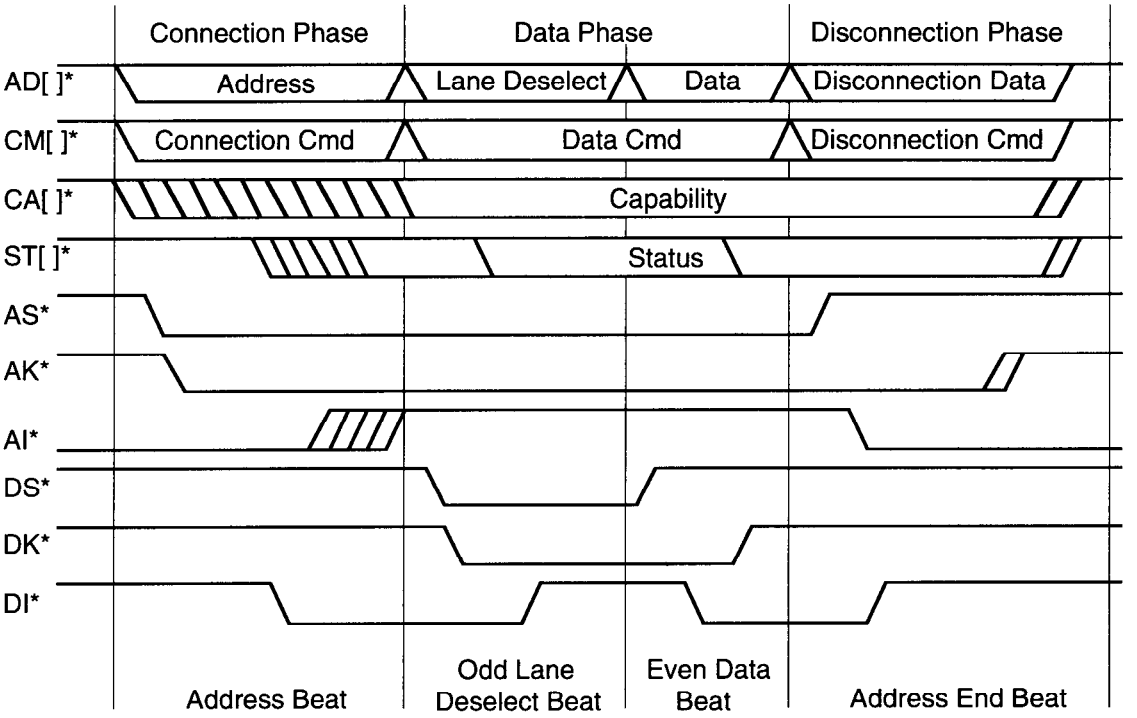


Figure 26—Compelled write partial transaction

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.19.9 Transaction error

In the previous example transactions, various status lines were introduced. **TE*** differs from all other status lines due to the fact that it is valid between transactions as shown in figure 27. A module asserts **te*** during the disconnection phase or data phase prior to releasing **ak***. The module maintains the assertion until it detects **AIf** at logic zero during the connection phase of the next transaction. This allows all bus modules to report errors no matter how late in a transaction they are detected. All bus modules can detect the state of **TE*** from the time they detect **AKf** at logic zero or **AS*** asserted until they release **ai***.

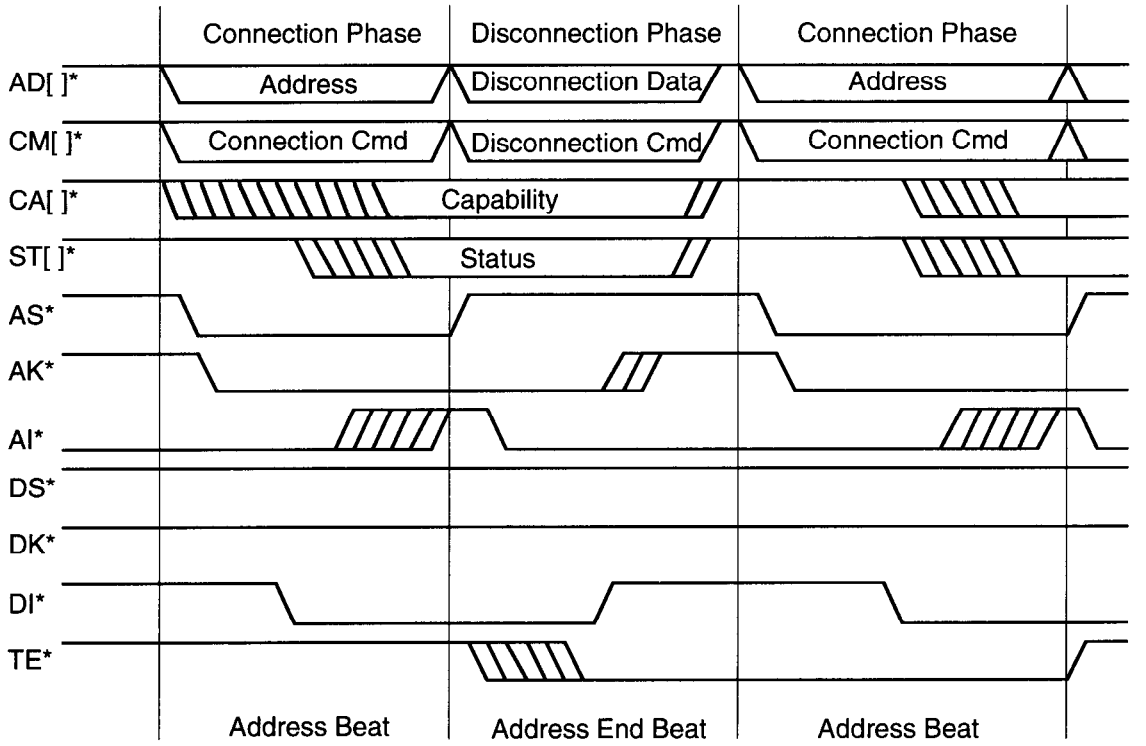


Figure 27—Transaction error timing

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.10 Packet data phase

A single odd packet of a packet data transfer phase is shown in figure 28. The connection and disconnection phase for a packet transaction is identical to that described for the compelled transfers. The figure illustrates the transfer of an eight-word packet (for example, a transfer of 64 bytes over a 64-bit ISO/IEC 10857 system). In an eight-word transfer there are 10-bit periods, each lasting a time period equal to the period of a local reference clock on the transmitting module.

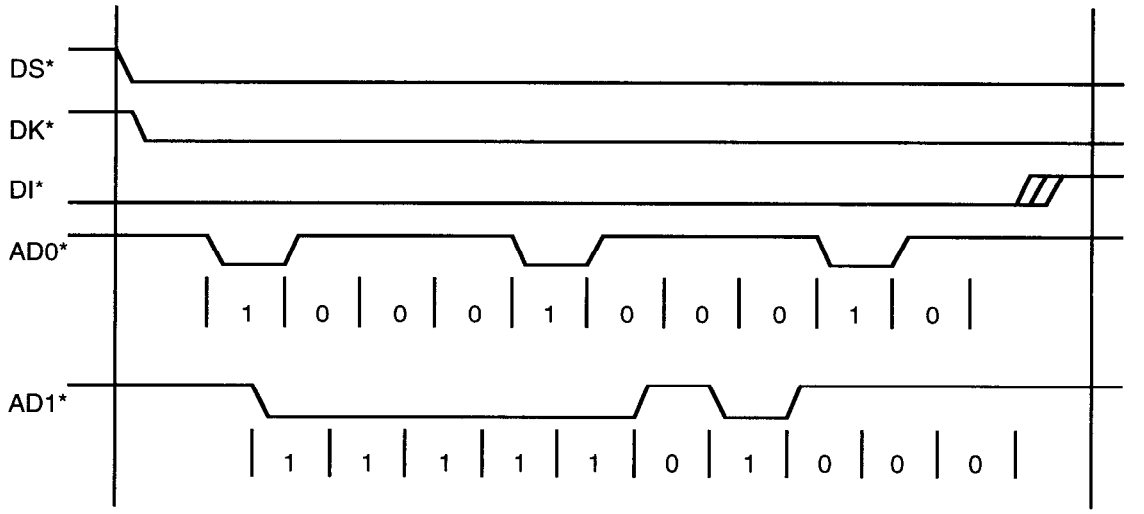


Figure 28—Packet data phase timing

The data transfer begins with the assertion of all of the address/data lines by the transmitting module. This transition defines the beginning of the first bit period. The receiving module synchronizes to each individual address/data bit using this initial transition. The transmitting module induces a transition in a bit period to indicate a "one". A bit period without a transition is a "zero". The first bit period is referred to as a synchronization bit. The last bit period is used for a longitudinal parity check. In the parity bit period, if the number of ones in the period bit periods is odd, the signal is a "one". If no longitudinal parity error occurs, all lines will be released at the end of the transmission.

For the packet mode, error signaling is provided only at the transaction level, using **TE***, so that speed can be sustained.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.19.11 Single packet read

In the single packet read transaction shown in figure 29, the slave sources a packet of data upon detecting the master's assertion of **DS*.** In this example, the slave delays the release of **di*** until the packet is completely transferred, although this is not a requirement. The master always waits until packets are completely transferred before releasing **as***, however. Unlike the compelled data protocol, the master is able to release **ds*** as soon as it detects the release of **di***, without first releasing **as***, as long as it doesn't assert **pr*** in the data command.

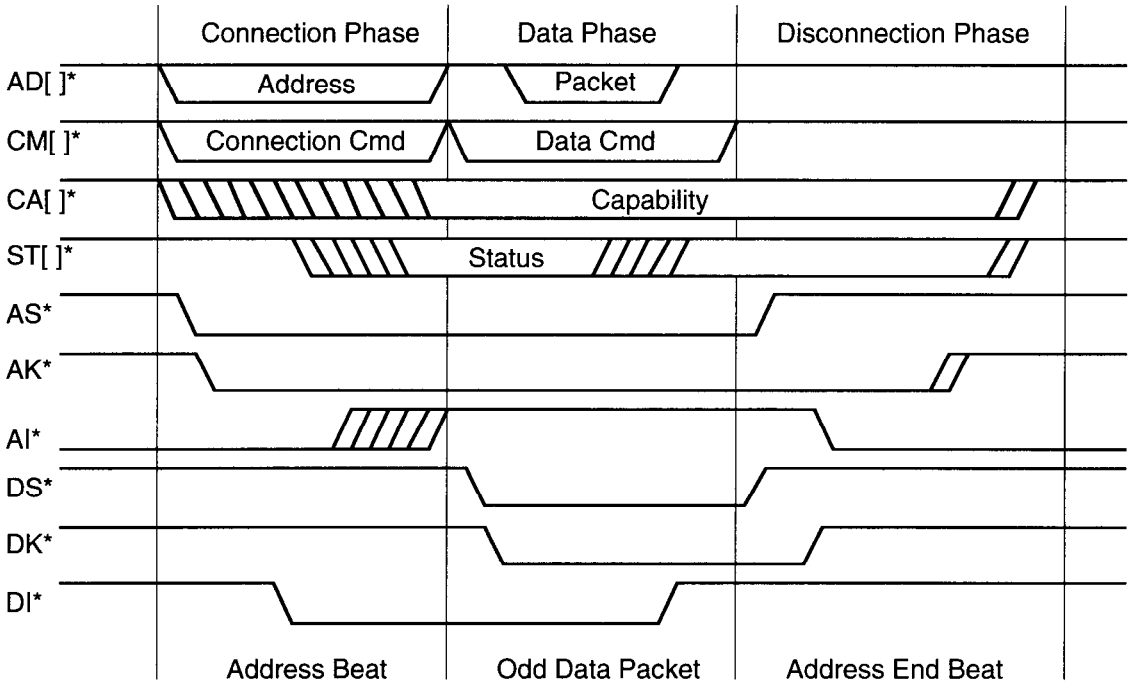


Figure 29—Single packet read transaction

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.12 Single packet read broadcast

Figure 30 shows a single packet read transaction with one or more slaves asserting the **bc*** status line. In this case, one or more slaves receive the packet along with the master.

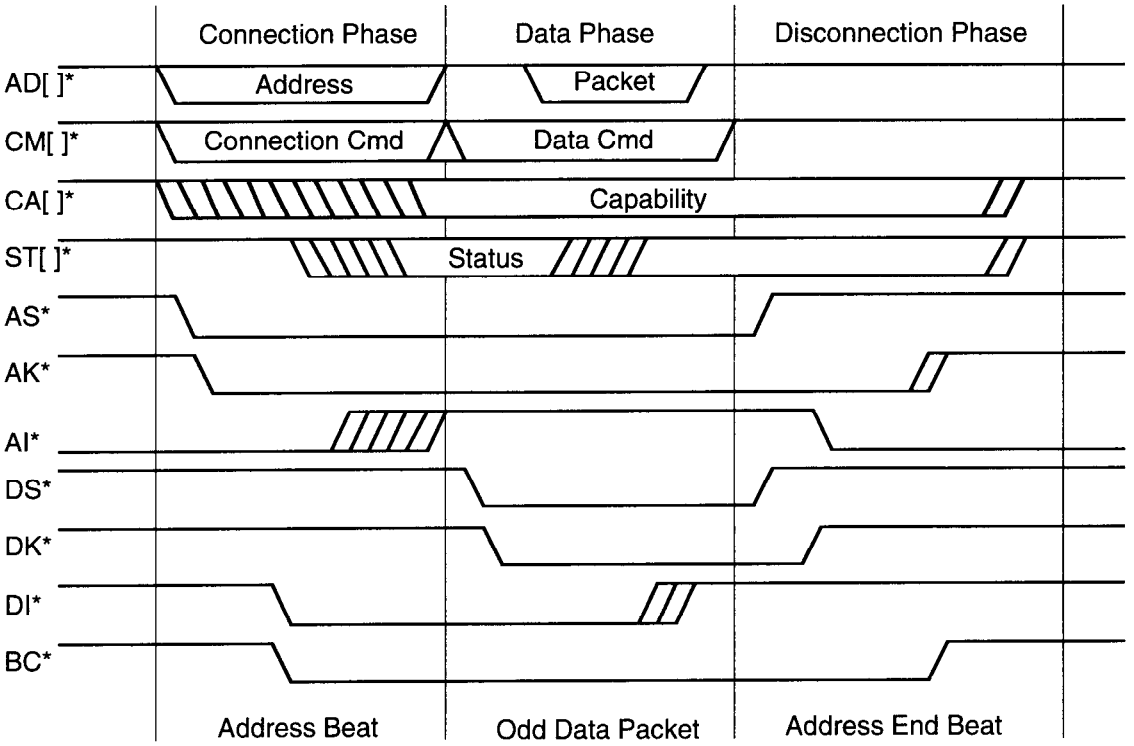


Figure 30—Single packet read transaction with broadcast

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.19.13 Single packet read intervention

The single packet read transaction with intervention, as shown in figure 31, is the same as the compelled version except that a packet is transferred.

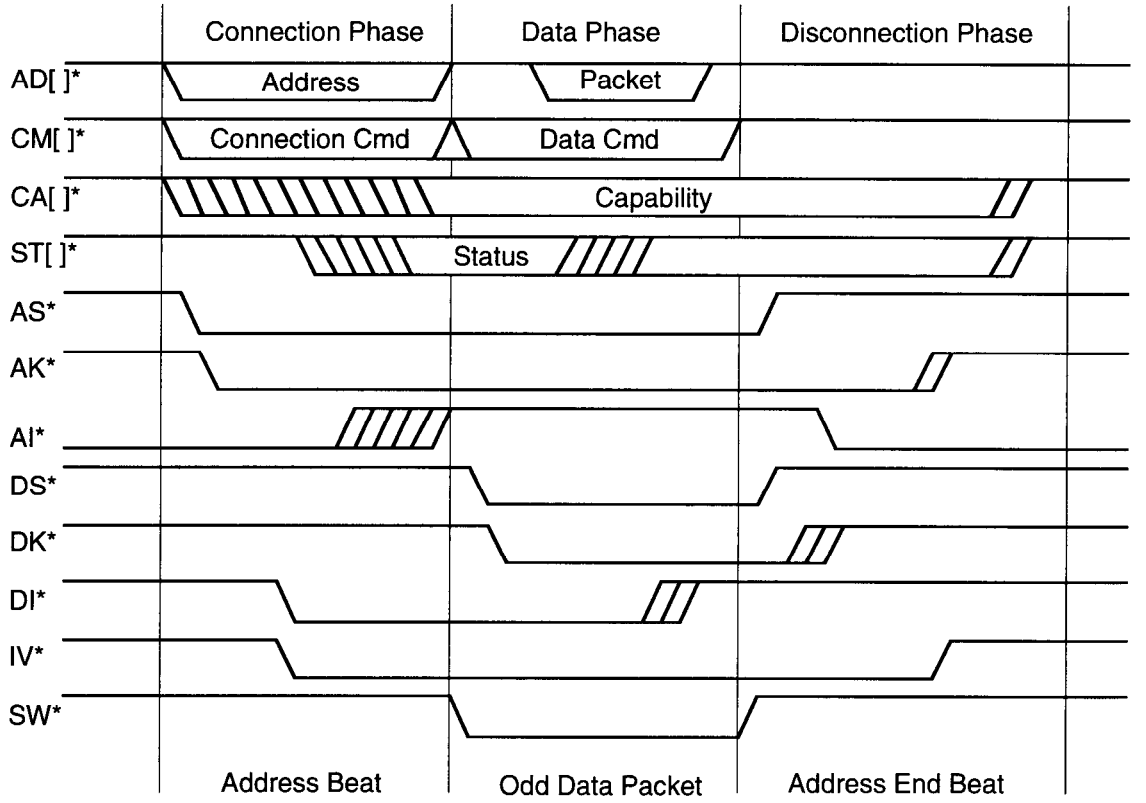


Figure 31—Single packet read transaction with intervention

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.19.14 Single packet write

A single packet write transaction, as shown in figure 32, is the same as the compelled version except that a packet is transferred. The master delays the transmission of the packet a length of time at least equal to one bit period after the assertion of **ds***. The slave must prepare its circuitry to receive the packet within this time period.

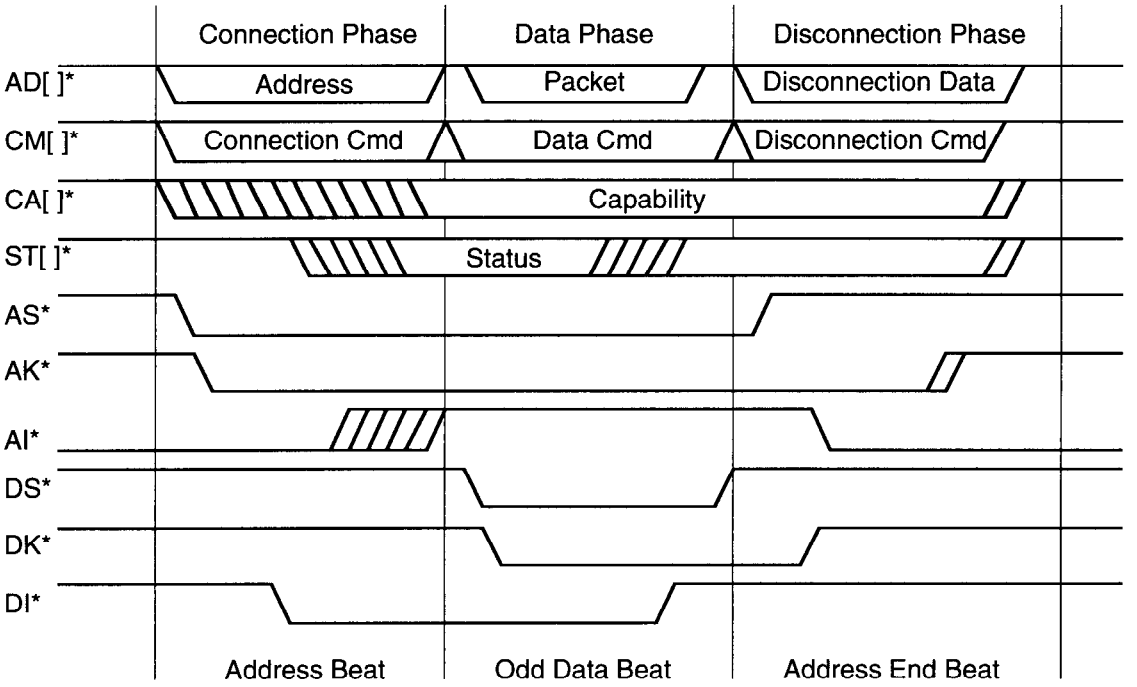


Figure 32—Single packet write transaction

6.2 Specification

6.2.1 Attribute definition

6.2.1.1 Transfer protocol attributes

MASTER. A module shall set MASTER if CENTRALIZED_MASTER | DISTRIBUTED_MASTER.

PACKET_ENABLED. The module shall set PACKET_ENABLED if PACKET_CAPABLE & (PACKET_LENGTH_2_ENABLE | PACKET_LENGTH_4_ENABLE | PACKET_LENGTH_8_ENABLE | PACKET_LENGTH_16_ENABLE | PACKET_LENGTH_32_ENABLE | PACKET_LENGTH_64_ENABLE).

COMPELLED. Modules may set COMPELLED if ~INIT & (MASTER & CONNECTION_PHASE | Participating). Modules shall set COMPELLED if ~INIT & (~PACKET_ENABLED & (MASTER & CONNECTION_PHASE | PARTICIPATING) | CONNECTION_PHASE & LOCKED). Modules shall maintain compelled set until INIT | TRANSACTION_COMPLETE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

COMPELLED_STATUS. Modules shall set COMPELLED_STATUS if ~INIT & **CA1*** & CONNECTION_STATUS. Modules shall maintain COMPELLED_STATUS set until INIT | TRANSACTION_COMPLETE.

LOW_SPEED. Modules shall set LOW_SPEED if ~INIT & ~HIGH_SPEED_ENABLE & (MASTER & CONNECTION_PHASE & ~COMPELLED | PARTICIPATING & ~COMPELLED). Modules shall maintain LOW_SPEED set until INIT | TRANSACTION_COMPLETE.

LOW_SPEED_STATUS. Modules shall set LOW_SPEED_STATUS if ~INIT & **CA0*** & CONNECTION_STATUS. Modules shall maintain LOW_SPEED_STATUS set until INIT | TRANSACTION_COMPLETE.

MULTIPLE_PACKET_MODE. Modules shall set MULTIPLE_PACKET_MODE if CONNECTION_PHASE & MULTIPLE_PACKET_MODE_ENABLE. Modules shall maintain MULTIPLE_PACKET_MODE set until INIT | TRANSACTION_COMPLETE.

PACKET_REQUEST. The MASTER shall set PACKET_REQUEST if ~INIT & MULTIPLE_PACKET_MODE prior to asserting **ds*** to begin the DATA_PHASE if it requests the transfer of more than one packet in the current transaction. The MASTER shall maintain PACKET_REQUEST set until INIT | TRANSFER_COUNT_ZERO & DATA_INFO_CAPTURED | TRANSACTION_COMPLETE.

Modules shall set PACKET_REQUEST if **CM0*** & PARTICIPATING & MULTIPLE_PACKET_MODE & DATA_INFO_CAPTURED. Modules shall maintain PACKET_REQUEST set until INIT | TRANSACTION_COMPLETE | **~CM0*** & DATA_INFO_CAPTURED.

6.2.1.2 Split transaction attributes

SPLIT. Modules may cause a split transaction by setting SPLIT if ~INIT & SPLIT_INITIATOR & SPLIT_ENABLE & CONNECTION_PHASE & (PARTICIPATING | MASTER & (SPLIT_CACHED | ~LOCK_HOLD) & ~(READ_LOCKED | READ_PARTIAL_LOCKED | READ_RESPONSE | WRITE_RESPONSE | COPYBACK | SHARED_RESPONSE). Modules shall maintain SPLIT set until INIT | TRANSACTION_COMPLETE.

SPLIT_STATUS. Modules shall set SPLIT_STATUS if ~INIT & **CA2*** & CONNECTION_STATUS. Modules shall maintain SPLIT_STATUS set until INIT | TRANSACTION_COMPLETE.

6.2.1.3 Address/data width attributes

32_BIT_CAPABLE. All modules shall be capable of participating in transactions with a 32-bit address and data path.

ADDR_WIDTH. The MASTER shall select a transaction with a 64-bit address width by setting ADDR_WIDTH if ~INIT & CONNECTION_PHASE & 64BIT_ADDRESS_CAPABLE & ~(READ_RESPONSE | WRITE_RESPONSE). The MASTER shall maintain ADDR_WIDTH set until INIT | TRANSACTION_COMPLETE.

64_BIT_ADDR. Modules shall set 64_BIT_ADDR if **CM7*** & PARTICIPATING & CONNECTION_PHASE | ADDR_WIDTH. Modules shall maintain 64_BIT_ADDR set until INIT | TRANSACTION_COMPLETE.

DATA_WIDTH_1. To select a 128-bit or 256-bit bus data width for a transaction the MASTER shall set DATA_WIDTH_1 if ~INIT & CONNECTION_PHASE. The MASTER shall maintain DATA_WIDTH_1 set until INIT | TRANSACTION_COMPLETE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

DATA_WIDTH_0. To select a 64-bit or 256-bit bus data width for a transaction the MASTER shall set DATA_WIDTH_0 if ~INIT & CONNECTION_PHASE. The MASTER shall maintain DATA_WIDTH_0 set until INIT | TRANSACTION_COMPLETE.

64_BIT_DATA. Modules shall set 64_bit_data if **~CM6*** & **CM5*** & PARTICIPATING & CONNECTION_PHASE | ~DATA_WIDTH_1 & DATA_WIDTH_0 and shall maintain it set until INIT | TRANSACTION_COMPLETE.

128_BIT_DATA. Modules shall set 128_BIT_DATA if **CM6*** & **~CM5*** & PARTICIPATING & CONNECTION_PHASE | DATA_WIDTH_1 & ~DATA_WIDTH_0 and shall maintain it set until INIT | TRANSACTION_COMPLETE.

256_BIT_DATA. Modules shall set 256_BIT_DATA if **CM6*** & **CM5*** & PARTICIPATING & CONNECTION_PHASE | DATA_WIDTH_1 & DATA_WIDTH_0 and shall maintain it set until INIT | TRANSACTION_COMPLETE.

6.2.1.4 Transaction attributes

READ_UNLOCKED. The MASTER shall set READ_UNLOCKED prior to the beginning of the CONNECTION_PHASE to perform a read unlocked transaction. Modules shall set READ_UNLOCKED if **~CM4*** & **~CM3*** & **~CM2*** & **~CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_UNLOCKED set until INIT | TRANSACTION_COMPLETE.

WRITE_UNLOCKED. The MASTER shall set WRITE_UNLOCKED prior to the beginning of the CONNECTION_PHASE to perform a write unlocked transaction. Modules shall set WRITE_UNLOCKED if **CM4*** & **~CM3*** & **~CM2*** & **~CM1*** & **~CM0*** & (PARTICIPATING | REQUEST_MAILBOX | RESPONSE_MAILBOX) & CONNECTION_PHASE. Modules shall maintain WRITE_UNLOCKED set until INIT | TRANSACTION_COMPLETE | DISCONNECTION_PHASE & ADDRESS_ONLY_UNLOCKED.

ADDRESS_ONLY_UNLOCKED. The MASTER shall set ADDRESS_ONLY_UNLOCKED prior to the beginning of the CONNECTION_PHASE to perform an address only unlocked transaction. Modules shall set ADDRESS_ONLY_UNLOCKED if **CM4*** & **~CM3*** & **~CM2*** & **~CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain ADDRESS_ONLY_UNLOCKED set until INIT | TRANSACTION_COMPLETE | DATA_PHASE.

READ_LOCKED. The MASTER shall set READ_LOCKED prior to the beginning of the CONNECTION_PHASE to perform a read locked transaction. Modules shall set READ_LOCKED if **~CM4*** & **~CM3*** & **~CM2*** & **~CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_LOCKED set until INIT | TRANSACTION_COMPLETE.

WRITE_LOCKED. The MASTER shall set WRITE_LOCKED prior to the beginning of the CONNECTION_PHASE to perform a write locked transaction. Modules shall set WRITE_LOCKED if **CM4*** & **~CM3*** & **~CM2*** & **~CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain WRITE_LOCKED set until INIT | TRANSACTION_COMPLETE | DISCONNECTION_PHASE & ADDRESS_ONLY_UNLOCKED.

ADDRESS_ONLY_LOCKED. The MASTER shall set ADDRESS_ONLY_LOCKED prior to the beginning of the CONNECTION_PHASE to perform an address only locked transaction. Modules shall set ADDRESS_ONLY_LOCKED if **CM4*** & **~CM3*** & **~CM2*** & **~CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain ADDRESS_ONLY_LOCKED set until INIT | TRANSACTION_COMPLETE | DATA_PHASE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

READ_PARTIAL. The MASTER shall set READ_PARTIAL prior to the beginning of the CONNECTION_PHASE to perform a read partial transaction. Modules shall set READ_PARTIAL if **~CM4*** & **~CM3*** & **~CM2*** & **CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_PARTIAL set until INIT | TRANSACTION_COMPLETE.

WRITE_PARTIAL. The MASTER shall set WRITE_PARTIAL prior to the beginning of the CONNECTION_PHASE to perform a write partial transaction. Modules shall set WRITE_PARTIAL if **CM4*** & **~CM3*** & **~CM2*** & **CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain WRITE_PARTIAL set until INIT | TRANSACTION_COMPLETE.

READ_PARTIAL_LOCKED. The MASTER shall set READ_PARTIAL_LOCKED prior to the beginning of the CONNECTION_PHASE to perform a read partial locked transaction. Modules shall set READ_PARTIAL_LOCKED if **~CM4*** & **~CM3*** & **~CM2*** & **CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_PARTIAL_LOCKED set until INIT | TRANSACTION_COMPLETE.

WRITE_PARTIAL_LOCKED. The MASTER shall set WRITE_PARTIAL_LOCKED prior to the beginning of the CONNECTION_PHASE to perform a write partial locked transaction. Modules shall set WRITE_PARTIAL_LOCKED if **CM4*** & **~CM3*** & **~CM2*** & **CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain WRITE_PARTIAL_LOCKED set until INIT | TRANSACTION_COMPLETE.

WRITE_RESPONSE. The MASTER shall set WRITE_RESPONSE prior to the beginning of the CONNECTION_PHASE to perform a write response transaction. Modules shall set WRITE_RESPONSE if **CM4*** & **~CM3*** & **CM2*** & **~CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain WRITE_RESPONSE set until INIT | TRANSACTION_COMPLETE.

READ_RESPONSE. The MASTER shall set READ_RESPONSE prior to the beginning of the CONNECTION_PHASE to perform a read response transaction. Modules shall set READ_RESPONSE if **CM4*** & **~CM3*** & **CM2*** & **~CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_RESPONSE set until INIT | TRANSACTION_COMPLETE.

WRITE_NO_ACKNOWLEDGE. The MASTER shall set WRITE_NO_ACKNOWLEDGE prior to the beginning of the CONNECTION_PHASE to perform a write no acknowledge transaction. Modules shall set WRITE_NO_ACKNOWLEDGE if **CM4*** & **~CM3*** & **CM2*** & **CM1*** & **~CM0*** & (PARTICIPATING | REQUEST_MAILBOX | RESPONSE_MAILBOX) & CONNECTION_PHASE. Modules shall maintain WRITE_NO_ACKNOWLEDGE set until INIT | TRANSACTION_COMPLETE.

READ_INVALID. The MASTER shall set READ_INVALID prior to the beginning of the CONNECTION_PHASE to perform a read invalid transaction. Modules shall set READ_INVALID if **~CM4*** & **CM3*** & **~CM2*** & **~CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_INVALID set until INIT | TRANSACTION_COMPLETE.

WRITE_INVALID. The MASTER shall set WRITE_INVALID prior to the beginning of the CONNECTION_PHASE to perform a write invalid transaction. Modules shall set WRITE_INVALID if **CM4*** & **CM3*** & **~CM2*** & **~CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain WRITE_INVALID set until INIT | TRANSACTION_COMPLETE.

READ_SHARED. The MASTER shall set READ_SHARED prior to the beginning of the CONNECTION_PHASE to perform a read shared transaction. Modules shall set READ_SHARED if **~CM4*** & **CM3*** & **~CM2*** & **~CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_SHARED set until INIT | TRANSACTION_COMPLETE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

COPY_BACK. The MASTER shall set COPY_BACK prior to the beginning of the CONNECTION_PHASE to perform a copy back transaction. Modules shall set COPY_BACK if **CM4*** & **CM3*** & **~CM2*** & **~CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain COPY_BACK set until INIT | TRANSACTION_COMPLETE.

READ_MODIFIED. The MASTER shall set READ_MODIFIED prior to the beginning of the CONNECTION_PHASE to perform a read modified transaction. Modules shall set READ_MODIFIED if **~CM4*** & **CM3*** & **~CM2*** & **CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain READ_MODIFIED set until INIT | TRANSACTION_COMPLETE.

INVALIDATE. The MASTER shall set INVALIDATE prior to the beginning of the CONNECTION_PHASE to perform an invalidate transaction. Modules shall set INVALIDATE if **CM4*** & **CM3*** & **~CM2*** & **CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain INVALIDATE set until INIT | TRANSACTION_COMPLETE.

SHARED_RESPONSE. The MASTER shall set SHARED_RESPONSE prior to the beginning of the CONNECTION_PHASE to perform a shared response transaction. Modules shall set SHARED_RESPONSE if **CM4*** & **CM3*** & **CM2*** & **~CM1*** & **~CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain SHARED_RESPONSE set until INIT | TRANSACTION_COMPLETE.

MODIFIED_RESPONSE. The MASTER shall set MODIFIED_RESPONSE prior to the beginning of the CONNECTION_PHASE to perform a modified response transaction. Modules shall set MODIFIED_RESPONSE if **CM4*** & **CM3*** & **CM2*** & **CM1*** & **CM0*** & PARTICIPATING & CONNECTION_PHASE. Modules shall maintain MODIFIED_RESPONSE set until INIT | TRANSACTION_COMPLETE.

CACHE_CMD. Modules shall set CACHE_CMD if READ_INVALID | WRITE_INVALID |READ_SHARED | COPY_BACK | READ_MODIFIED | INVALIDATE | SHARED_RESPONSE | MODIFIED_RESPONSE.

WRITE_CMD. The MASTER shall set WRITE_CMD if WRITE_UNLOCKED | ADDRESS_ONLY_UNLOCKED | WRITE_LOCKED | ADDRESS_ONLY_LOCKED | WRITE_PARTIAL | WRITE_PARTIAL_LOCKED | WRITE_RESPONSE | READ_RESPONSE | WRITE_NO_ACKNOWLEDGE | WRITE_INVALID | COPY_BACK | INVALIDATE | SHARED_RESPONSE | MODIFIED_RESPONSE.

TRANS_CMD_3. The MASTER shall set TRANS_CMD_3 if READ_INVALID | WRITE_INVALID | READ_SHARED | COPY_BACK | READ_MODIFIED | INVALIDATE | SHARED_RESPONSE | MODIFIED_RESPONSE.

TRANS_CMD_2. The MASTER shall set TRANS_CMD_2 if WRITE_RESPONSE | READ_RESPONSE | WRITE_NO_ACKNOWLEDGE | SHARED_RESPONSE | MODIFIED_RESPONSE.

TRANS_CMD_1. The MASTER shall set TRANS_CMD_1 if READ_PARTIAL | WRITE_PARTIAL | READ_PARTIAL_LOCKED | WRITE_PARTIAL_LOCKED | WRITE_NO_ACKNOWLEDGE | READ_MODIFIED | INVALIDATE | MODIFIED_RESPONSE.

TRANS_CMD_0. The MASTER shall set TRANS_CMD_0 if READ_LOCKED | WRITE_LOCKED | ADDRESS_ONLY_LOCKED | READ_PARTIAL_LOCKED | WRITE_PARTIAL_LOCKED | READ_RESPONSE | READ_SHARED | COPY_BACK | READ_MODIFIED | INVALIDATE | MODIFIED_RESPONSE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

PARTIAL. Modules shall set PARTIAL if READ_PARTIAL | WRITE_PARTIAL | READ_PARTIAL_LOCKED | WRITE_PARTIAL_LOCKED.

ADDRESS_ONLY. Modules shall set ADDRESS_ONLY if ~INIT & CONNECTION_PHASE and shall maintain it set until INIT | DATA_PHASE | TRANSACTION_COMPLETE.

SLAVE_WRITE. Modules shall set SLAVE_WRITE if **CM4*** validated by the transition of DS* and maintain it set until INIT | TRANSACTION_COMPLETE | MULTIPLE_PACKET_MODE & **~CM4*** & DATA_INFO_CAPTURED.

WRITE_STATUS. Modules shall set WRITE_STATUS if ~INIT & CONNECTION_PHASE & **CM4*** and shall maintain it set until INIT | TRANSACTION_COMPLETE.

MORE. The MASTER may set MORE at the beginning of a block copy operation. The MASTER shall clear MORE before or on the last transaction of a block copy operation.

LAST_TRANSACTION_FLAG. Modules shall set LAST_TRANSACTION_FLAG if DATA_INFO_CAPTURED & MULTIPLE_PACKET_MODE & **CM1*** and maintain it set until INIT | TRANSACTION_COMPLETE | DATA_INFO_CAPTURED & **~CM1***.

6.2.1.5 Locked operations/attributes

LOCKED. Modules shall set LOCKED if READ_LOCKED | WRITE_LOCKED | ADDRESS_ONLY_LOCKED | READ_PARTIAL_LOCKED | WRITE_PARTIAL_LOCKED.

RESOURCE_LOCK. A module shall maintain a RESOURCE_LOCK attribute for each separately lockable resource it contains associated with particular addresses for which it will set SELECTED. Modules shall set RESOURCE_LOCK if LOCKED & SELECTED and shall maintain RESOURCE_LOCK set until INIT | ~LOCKED & ~LOCK_HOLD | **~ET*** & **~AS***.

LOCK_HOLD. The MASTER shall set LOCK_HOLD if RESOURCE_LOCK prior to the beginning of the DISCONNECTION_PHASE to cause modules RESOURCE_LOCK attributes to remain set past the end of the current transaction. Modules shall set LOCK_HOLD if SELECTED & **CM0*** and they detect the release of **AS***. Modules shall maintain LOCK_HOLD set until INIT | **~ET*** | ~LOCKED & **AS*** & **~ai*.**

MASK_SWAP. To select a MASK_SWAP locked command the MASTER shall set MASK_SWAP if WRITE_LOCKED | WRITE_PARTIAL_LOCKED prior to the beginning of the DATA_PHASE. Modules shall set MASK_SWAP if **~CM2*** & **~CM1*** & **CM0*** & PARTICIPATING & DATA_PHASE & RESOURCE_LOCK. Modules shall maintain MASK_SWAP set until INIT | TRANSACTION_COMPLETE.

COMPARE_SWAP. To select a COMPARE_SWAP locked command the MASTER shall set COMPARE_SWAP if WRITE_LOCKED | WRITE_PARTIAL_LOCKED prior to the beginning of the DATA_PHASE. Modules shall set COMPARE_SWAP if **~CM2*** & **CM1*** & **~CM0*** & PARTICIPATING & DATA_PHASE & RESOURCE_LOCK. Modules shall maintain COMPARE_SWAP set until INIT | TRANSACTION_COMPLETE.

FETCH_ADD_BIG. To select a FETCH_ADD_BIG locked command the MASTER shall set FETCH_ADD_BIG if WRITE_LOCKED | WRITE_PARTIAL_LOCKED prior to the beginning of the DATA_PHASE. Modules shall set FETCH_ADD_BIG if **~CM2*** & **CM1*** & **CM0*** & PARTICIPATING & DATA_PHASE & RESOURCE_LOCK. Modules shall maintain FETCH_ADD_BIG set until INIT | TRANSACTION_COMPLETE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

FETCH_ADD_LITTLE. To select a FETCH_ADD_LITTLE locked command the master shall set FETCH_ADD_LITTLE if WRITE_LOCKED | WRITE_PARTIAL_LOCKED prior to the beginning of the DATA_PHASE. Modules shall set FETCH_ADD_LITTLE if **CM2*** & **~CM1*** & **~CM0*** & PARTICIPATING & DATA_PHASE & RESOURCE_LOCK. Modules shall maintain FETCH_ADD_LITTLE set until INIT | TRANSACTION_COMPLETE.

LOCK_CMD_0. The MASTER shall set LOCK_CMD_0 if MASK_SWAP | FETCH_ADD_BIG.

LOCK_CMD_1. The MASTER shall set LOCK_CMD_1 if COMPARE_SWAP | FETCH_ADD_BIG.

LOCK_CMD_2. The MASTER shall set LOCK_CMD_2 if FETCH_ADD_LITTLE.

6.2.1.6 Data transfer length

TRANSFER_UNRESTRICTED. To specify an unrestricted transfer length the MASTER shall set TRANSFER_UNRESTRICTED prior to the DATA_PHASE | DISCONNECTION_PHASE. The MASTER shall set TRANSFER_UNRESTRICTED if LOCK_CMD_0 | LOCK_CMD_1 | LOCK_CMD_2. The master shall only set TRANSFER_UNRESTRICTED if COMPELLED_STATUS | ~(SPLIT_STATUS & ~WRITE_CMD). The MASTER shall not set TRANSFER_UNRESTRICTED and shall set one of TRANSFER_1 | TRANSFER_2 | TRANSFER_4 | TRANSFER_8 | TRANSFER_16 | TRANSFER_32 | TRANSFER_64 if ~COMPELLED_STATUS | SPLIT_STATUS & ~WRITE_CMD. Modules shall set TRANSFER_UNRESTRICTED if ~INIT & PARTICIPATING & DATA_PHASE & **~CM7*** & **~CM6*** & **~CM5***. Modules shall maintain TRANSFER_UNRESTRICTED set until INIT | TRANSACTION_COMPLETE.

TRANSFER_1. To specify a transfer length of one the MASTER shall set TRANSFER_1 prior to the DATA_PHASE | DISCONNECTION_PHASE. The MASTER shall not set TRANSFER_1 if ~COMPELLED_STATUS. Modules shall set TRANSFER_1 if ~INIT & PARTICIPATING & DATA_PHASE & **~CM7*** & **~CM6*** & **CM5***. Modules shall maintain TRANSFER_1 set until INIT | TRANSACTION_COMPLETE.

TRANSFER_2. To specify a transfer length of two the MASTER shall set TRANSFER_2 prior to the DATA_PHASE | DISCONNECTION_PHASE. Modules shall set TRANSFER_2 if ~INIT & PARTICIPATING & DATA_PHASE & **~CM7*** & **CM6*** & **~CM5***. Modules shall maintain TRANSFER_2 set until INIT | TRANSACTION_COMPLETE.

TRANSFER_4. To specify a transfer length of four the MASTER shall set TRANSFER_4 prior to the DATA_PHASE | DISCONNECTION_PHASE. Modules shall set TRANSFER_4 if ~INIT & PARTICIPATING & DATA_PHASE & **~CM7*** & **CM6*** & **CM5***. Modules shall maintain TRANSFER_4 set until INIT | TRANSACTION_COMPLETE.

TRANSFER_8. To specify a transfer length of eight the MASTER shall set TRANSFER_8 prior to the DATA_PHASE | DISCONNECTION_PHASE. Modules shall set TRANSFER_8 if ~INIT & PARTICIPATING & DATA_PHASE & **CM7*** & **~CM6*** & **~CM5***. Modules shall maintain TRANSFER_8 set until INIT | TRANSACTION_COMPLETE.

TRANSFER_16. To specify a transfer length of 16 the MASTER shall set TRANSFER_16 prior to the DATA_PHASE | DISCONNECTION_PHASE. Modules shall set TRANSFER_16 if ~INIT & PARTICIPATING & DATA_PHASE & **CM7*** & **~CM6*** & **CM5***. Modules shall maintain TRANSFER_16 set until INIT | TRANSACTION_COMPLETE.

TRANSFER_32. To specify a transfer length of 32 the MASTER shall set TRANSFER_32 prior to the DATA_PHASE | DISCONNECTION_PHASE. Modules shall set TRANSFER_32 if ~INIT &

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

PARTICIPATING & DATA_PHASE & **CM7*** & **CM6*** & **~CM5***. Modules shall maintain TRANSFER_32 set until INIT | TRANSACTION_COMPLETE.

TRANSFER_64. To specify a transfer length of 64 the MASTER shall set TRANSFER_64 prior to the DATA_PHASE | DISCONNECTION_PHASE. Modules shall set TRANSFER_64 if ~INIT & PARTICIPATING & DATA_PHASE & **CM7*** & **CM6*** & **CM5***. Modules shall maintain TRANSFER_64 set until INIT | TRANSACTION_COMPLETE.

DATA_LENGTH_2. The MASTER shall set DATA_LENGTH_2 if TRANSFER_8 | TRANSFER_16 | TRANSFER_32 | TRANSFER_64.

DATA_LENGTH_1. The MASTER shall set DATA_LENGTH_1 if TRANSFER_2 | TRANSFER_4 | TRANSFER_32 | TRANSFER_64.

DATA_LENGTH_0. The MASTER shall set DATA_LENGTH_0 if TRANSFER_1 | TRANSFER_4 | TRANSFER_16 | TRANSFER_64.

6.2.1.7 Transaction status

6.2.1.7.1 Master transaction status

BUSY_STATUS. The MASTER shall set BUSY_STATUS if **ST1*** and it detects the transition of **AIf** to logic zero. The MASTER shall maintain BUSY_STATUS set until INIT | TRANSACTION_COMPLETE.

SELECTED_STATUS. The MASTER shall set SELECTED_STATUS if **ST2*** and it detects the transition of **AIf** to logic zero. The MASTER shall maintain SELECTED_STATUS set until INIT | TRANSACTION_COMPLETE | MULTIPLE_PACKET_MODE & **~ST2*** and it detects the transition of **DKf** or **DIf** to logic zero.

TRANSACTION_FLAG_STATUS. The MASTER shall set TRANSACTION_FLAG_STATUS if **ST4*** & (CONNECTION_INFO_CAPTURED & ~DATA_PHASE | DATA_INFO_CAPTURED & MULTIPLE_PACKET_MODE). The MASTER shall maintain TRANSACTION_FLAG_STATUS set until INIT | TRANSACTION_COMPLETE | **~ST4*** & MULTIPLE_PACKET_MODE & DATA_INFO_CAPTURED.

INTERVENTION_STATUS. The MASTER shall set INTERVENTION_STATUS if **ST5*** & (CONNECTION_INFO_CAPTURED & ~DATA_PHASE | DATA_INFO_CAPTURED & MULTIPLE_PACKET_MODE). The MASTER shall maintain INTERVENTION_STATUS set until INIT | TRANSACTION_COMPLETE | **~ST5*** & MULTIPLE_PACKET_MODE & DATA_INFO_CAPTURED.

BEAT_ERROR_STATUS. The MASTER shall set BEAT_ERROR_STATUS if **ST6*** & CONNECTION_INFO_CAPTURED and shall maintain it set until INIT | TRANSACTION_COMPLETE.

WAIT_STATUS. The MASTER shall set WAIT_STATUS if **ST7*** & (CONNECTION_INFO_CAPTURED | DATA_INFO_CAPTURED & MULTIPLE_PACKET_MODE) and shall maintain it set until INIT | TRANSACTION_COMPLETE.

END_OF_DATA_STATUS. The MASTER shall set END_OF_DATA_STATUS if **ST1*** & DATA_INFO_CAPTURED. The MASTER shall maintain END_OF_DATA_STATUS set until INIT | TRANSACTION_COMPLETE.

Modules shall ignore data during compelled transactions for beats subsequent to that in which END_OF_DATA is set.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

MASTER_CAPABILITY_ERROR. The MASTER shall set MASTER_CAPABILITY_ERROR if one or more slaves signal the use of a facility that the master does not support. The MASTER shall maintain MASTER_CAPABILITY_ERROR set until INIT | TRANSACTION_COMPLETE.

6.2.1.7.2 Module transaction status

ADDRESS_PARITY_ERROR. Modules shall set ADDRESS_PARITY_ERROR if a parity error is detected during the connection phase on the **AD[ ]*** lines. Modules shall maintain ADDRESS_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

BROADCAST_STATUS. Modules shall set BROADCAST_STATUS if MASTER & (**ST3*** | **~CA1*** & MULTIPLE_PACKET_MODE) and it detects the transition of **AIf** to logic zero, or PARTICIPATING & **ST3*** and it detects the first assertion of **DS***. Modules shall maintain BROADCAST_STATUS set until INIT | TRANSACTION_COMPLETE.

DATA_PARITY_ERROR. PARTICIPATING modules shall set DATA_PARITY_ERROR if a parity error is detected during the data phase on the **AD[ ]*** or **D[ ]*** lines. DATA_PARITY_ERROR shall also be set if a longitudinal parity error is detected during packet mode. Modules shall maintain DATA_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

DISCONNECTION_DATA_PARITY_ERROR. Modules shall set DISCONNECTION_DATA_PARITY_ERROR if a parity error is detected during the disconnection phase on **AD[31..0]*** when WRITE_CMD | SPLIT_STATUS. Modules shall maintain DISCONNECTION_DATA_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

CONNECTION_COMMAND_PARITY_ERROR. Modules shall set CONNECTION_COMMAND_PARITY_ERROR if a parity error is detected during the connection phase on the **CM[ ]*** lines. Modules shall maintain CONNECTION_COMMAND_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

DATA_COMMAND_PARITY_ERROR. PARTICIPATING modules shall check parity on the command lines during each data beat. PARTICIPATING modules shall set DATA_COMMAND_PARITY_ERROR if a parity error is detected during the data phase on the **CM[ ]*** lines. Modules shall maintain DATA_COMMAND_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

DISCONNECTION_COMMAND_PARITY_ERROR. Modules shall set DISCONNECTION_COMMAND_PARITY_ERROR if a parity error is detected during the disconnection phase on the **CM[ ]*** lines. Modules shall maintain DISCONNECTION_COMMAND_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

CAPABILITY_ERROR. PARTICIPATING modules shall set CAPABILITY_ERROR if the master requests an operation during the connection phase or data phase which the module is incapable of performing. Modules shall maintain CAPABILITY_ERROR set until INIT | TRANSACTION_COMPLETE.

ILLEGAL_OPERATION_ERROR. PARTICIPATING modules shall set ILLEGAL_OPERATION_ERROR if a module detects a combination of command, status, and/or capability that results in an illegal operation. Modules shall maintain ILLEGAL_OPERATION_ERROR set until INIT | TRANSACTION_COMPLETE.

ADDRESS_TAG_PARITY_ERROR. Modules shall set ADDRESS_TAG_PARITY_ERROR if a parity error is detected during the connection phase on the **TG[ ]*** lines and these lines are activated. Modules shall maintain ADDRESS_TAG_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

DATA_TAG_PARITY_ERROR. PARTICIPATING modules shall set DATA_TAG_PARITY_ERROR if a parity error is detected during the data phase on the **TG[ ]*** lines and these lines are activated. Modules shall maintain DATA_TAG_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

DISCONNECTION_TAG_PARITY_ERROR. Modules shall set DISCONNECTION_TAG_PARITY_ERROR if a parity error is detected during the disconnection phase on the **TG[ ]*** lines and these lines are activated. Modules shall maintain DISCONNECTION_TAG_PARITY_ERROR set until INIT | TRANSACTION_COMPLETE.

BEAT_ERROR. Modules shall set BEAT_ERROR if PARITY_REPORT_ENABLE & (ADDRESS_PARITY_ERROR | CONNECTION_COMMAND_PARITY_ERROR | ADDRESS_TAG_PARITY_ERROR) | CAPABILITY_ERROR | ILLEGAL_OPERATION_ERROR. Modules should use BEAT_ERROR only for precise error reporting. Modules may set BEAT_ERROR if PARITY_REPORT_ENABLE & (DATA_PARITY_ERROR | DATA_COMMAND_PARITY_ERROR | DATA_TAG_PARITY_ERROR). Modules shall maintain BEAT_ERROR set until INIT | TRANSACTION_COMPLETE.

TRANSACTION_ERROR. Modules shall set TRANSACTION_ERROR if BEAT_ERROR | PARITY_REPORT_ENABLE & (DATA_PARITY_ERROR | DISCONNECTION_DATA_PARITY_ERROR | DATA_COMMAND_PARITY_ERROR | DISCONNECTION_COMMAND_PARITY_ERROR | DATA_TAG_PARITY_ERROR | DISCONNECTION_TAG_PARITY_ERROR).

TRANSACTION_ERROR_STATUS. Modules shall set TRANSACTION_ERROR_STATUS if **ST0*** & DISCONNECTION_END. Modules shall clear TRANSACTION_ERROR_STATUS if INIT | TRANSACTION_COMPLETE.

BUSY. A module shall set BUSY if ADDR_DECODED & CONNECTION_PHASE & PARTICIPATING & (MESSAGE_BUSY or the module can not perform the transaction until a later time and it wants the master to retry later). Modules shall maintain BUSY set until INIT | TRANSACTION_COMPLETE.

SELECTED. Modules shall set SELECTED if ~(SHARED_RESPONSE | MODIFIED_RESPONSE) & ADDR_DECODED and the address corresponding to the next data beat is one for which the module is the ultimate source and/or destination of the data or an agent acting on behalf of such a module, or if REQUESTER & (SHARED_RESPONSE | MODIFIED_RESPONSE) & ADDR_DECODED and the address corresponding to the next data beat matches the cache line address that is associated with this substantiation of REQUESTER, or if MAILBOX_SELECTED.

Modules shall maintain SELECTED set until INIT | TRANSACTION_COMPLETE | ADDR_DECODED and the address corresponding to the next data beat is one for which the conditions in the previous paragraph are not true.

BROADCAST. Modules may set BROADCAST if ~INIT & ADDR_DECODED & **ai***. Modules shall maintain BROADCAST set until INIT | TRANSACTION_COMPLETE.

BROADCALL_ADDRESS. Modules shall set BROADCALL_ADDRESS if ~INIT & ADDR_DECODED & ~WRITE_STATUS & **ai*** and the address corresponding to the next data beat is one it should respond to in addition to other slaves in broadcall mode. Parity on the address/data lines shall not be checked during the data phase if BROADCALL_ADDRESS is set.

WAIT. A module shall set WAIT if ~INIT & ADDR_DECODED & (WAIT_CACHED or the module cannot perform the transaction until a later time), and shall perform a system defined action to notify the requesting module it is ready to perform the transaction. Modules shall maintain WAIT set until INIT | TRANSACTION_COMPLETE.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

END_OF_DATA. Modules shall set END_OF_DATA if COMPELLED_STATUS & DATA_PHASE & ADDR_DECODED and the address corresponding to the next data beat is one for which the module is unable to supply or receive data & ~(PARTIAL and the first beat of the DATA_PHASE). Modules shall maintain END_OF_DATA set until INIT | TRANSACTION_COMPLETE.

PARTICIPATING. A module shall set PARTICIPATING if SELECTED | INTERVENTION | BROADCAST | MULTIPLE_PACKET_MODE & CACHE & CACHE_COMMAND & ~COMPELLED_STATUS.

PACKET_NAK. Modules shall set PACKET_NAK if MULTIPLE_PACKET_MODE & PACKET_REQUEST and the module chooses to prevent the transfer of the requested packet. The module shall prevent the transfer of the requested packet if it can no longer be the source of data or it wishes to become the source of data. Modules shall maintain PACKET_NAK set until INIT | TRANSACTION_COMPLETE.

6.2.1.8 Transaction timing attributes

TRANSACTION_BEGIN. The MASTER shall set TRANSACTION_BEGIN when it desires to perform a transaction and shall maintain it set until DATA_PHASE | DISCONNECTION_PHASE during its last transaction.

TRANSACTION_TIMEOUT. The MASTER shall clear TRANSACTION_TIMEOUT at the beginning of each transaction.

The attribute TRANSACTION_TIMEOUT is such that 128 μs ± 30% after it is cleared it shall set itself if TRANSACTION_COMPLETE has not set. The duration of TRANSACTION_TIMEOUT may be changed as specified in 7.2.4.

TRANSACTION_COMPLETE. Modules shall set TRANSACTION_COMPLETE if INIT | DISCONNECTION_END and they have sampled **TE*** and have updated the state of any internal variables based upon the completion of the transaction. The MASTER shall clear TRANSACTION_COMPLETE if **as*** & **ak***. Modules shall clear TRANSACTION_COMPLETE if **~ai***.

ADDR_DECODED. Modules shall maintain facilities to decode the address of the next data (not lane deselect) beat. This address shall initially be set to the address received on the **AD[ ]*** lines and validated by the assertion of **AS***. This address shall be subsequently updated by each PARTICIPATING module after each assertion or release of **DS*** for which data is transferred. The method by which this address is updated is system dependent and is not specified, except that it shall correspond to the address of the data which would be transferred in the next data beat. A module shall set ADDR_DECODED when it has decoded the address corresponding to the next data beat and determined its response to it. During the connection phase, if ADDRESS_PARITY_ERROR | CONNECTION_COMMAND_PARITY_ERROR, the module shall not set ADDR_DECODED for the duration of the transaction. Modules shall maintain ADDR_DECODED set until the module releases **ai*** | **dk*** | **di***.

CONNECTION_PHASE. The MASTER shall set CONNECTION_PHASE if ~INIT & ~BUS_HOLD & TRANSACTION_BEGIN & **~AKf** & **AI***. Modules shall set CONNECTION_PHASE if **AS*** & **ai***. Modules shall maintain CONNECTION_PHASE set until INIT | DATA_PHASE | DISCONNECTION_PHASE.

DATA_PHASE. The MASTER shall set DATA_PHASE when ~INIT & ~TRANSFER_COUNT_ZERO & **as*** & CONNECTION_INFO_CAPTURED and sufficient time has been allowed for TRANSACTION_FLAG_STATUS & INTERVENTION_STATUS to have stabilized. Modules shall set DATA_PHASE when **AS*** & **DS*** & CONNECTION_INFO_CAPTURED. Modules shall maintain DATA_PHASE set until INIT | DISCONNECTION_PHASE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

DISCONNECTION_PHASE. The MASTER shall set DISCONNECTION_PHASE when ~INIT & TRANSFER_COUNT_ZERO & CONNECTION_INFO_CAPTURED. Modules shall set DISCONNECTION_PHASE when they detect the release of **AS*.** Modules shall maintain DISCONNECTION_PHASE set until INIT | DISCONNECTION_END.

CONNECTION_INFO_CAPTURED. Modules shall set CONNECTION_INFO_CAPTURED if CONNECTION_STATUS and they have captured the appropriate connection phase information. Modules shall maintain CONNECTION_INFO_CAPTURED set until INIT | DISCONNECTION_END.

DATA_INFO_CAPTURED. The MASTER shall set DATA_INFO_CAPTURED if DATA_PHASE and it has captured the slave information, and

1) ~BROADCAST_STATUS & (**ds*** & **~DI*** | **~ds*** & **~DK***).
2) BROADCAST_STATUS & (**ds*** & **~DIf** | **~ds*** & **~DKf**).

The MASTER shall maintain DATA_INFO_CAPTURED set until it causes the next transition of **ds*** or until DISCONNECTION_PHASE.

If ~(SLAVE_WRITE | BROADCAST) a PARTICIPATING slave shall set DATA_INFO_CAPTURED when it has captured the MASTER information validated by a transition of **DS***.

If SLAVE_WRITE | BROADCAST a PARTICIPATING slave shall set DATA_INFO_CAPTURED when it has captured the information validated by (**DS*** & **DK*** | **~DS*** & **DI***). Slaves shall guarantee that the delays through their **DS*** logic are greater than or equal to the delays through their **DK*** or **DI*** logic with respect to the requirement of this paragraph.

PARTICIPATING slaves shall maintain DATA_INFO_CAPTURED set until they release **dk*** or **di***.

DISC_INFO_CAPTURED. A PARTICIPATING slave shall set DISC_INFO_CAPTURED when it has captured the MASTER information validated by the release of **AS*** and shall maintain DISC_INFO_CAPTURED set until it releases **ak***.

TRANSFER_COUNT_ZERO. The MASTER shall clear TRANSFER_COUNT_ZERO prior to the beginning of the CONNECTION_PHASE if it plans to transfer data during the DATA_PHASE.

Immediately after the MASTER has caused a transition of **ds*** requesting the last beat or packet of the transaction it shall set TRANSFER_COUNT_ZERO.

The MASTER shall set TRANSFER_COUNT_ZERO if COMPELLED_STATUS & TRANSFER_1 & ~PARTIAL and it has asserted **ds***.

The MASTER shall set TRANSFER_COUNT_ZERO if COMPELLED_STATUS & TRANSFER_1 & PARTIAL and it has released **ds***.

The MASTER shall set TRANSFER_COUNT_ZERO if COMPELLED_STATUS and it has initiated the data beat by a transition of **ds*** whose low order address bits, represented by **A[ ]***, correspond to one of the following rules:

If ~DATA_WIDTH_1 & ~DATA_WIDTH_0, then set N=2
If ~DATA_WIDTH_1 & DATA_WIDTH_0, then set N=3
If DATA_WIDTH_1 & ~DATA_WIDTH_0, then set N=4
If DATA_WIDTH_1 & DATA_WIDTH_0, then set N=5
If TRANSFER_2 and **A[N]*** equals 1, set TRANSFER_COUNT_ZERO
If TRANSFER_4 and **A[N+1..N]*** both equal 1, set TRANSFER_COUNT_ZERO

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If TRANSFER_8 and **A[N+2..N]*** all equal 1, set TRANSFER_COUNT_ZERO
If TRANSFER_16 and **A[N+3..N]*** all equal 1, set TRANSFER_COUNT_ZERO
If TRANSFER_32 and **A[N+4..N]*** all equal 1, set TRANSFER_COUNT_ZERO
If TRANSFER_64 and **A[N+5..N]*** all equal 1, set TRANSFER_COUNT_ZERO

The MASTER shall set TRANSFER_COUNT_ZERO if (**ST6*** | **~ST2***) & CONNECTION_INFO_CAPTURED | TRANSACTION_ERROR | WAIT_STATUS | BUSY_STATUS | ADDRESS_ONLY_UNLOCKED | ADDRESS_ONLY_LOCKED | INVALIDATE | WRITE_RESPONSE | MASTER_CAPABILITY_ERROR.

The MASTER shall set TRANSFER_COUNT_ZERO if SPLIT_STATUS & (READ_LOCKED | READ_UNLOCKED | READ_INVALID | READ_SHARED | READ_MODIFIED | COPY_BACK | (READ_PARTIAL | READ_PARTIAL_LOCKED) & it has asserted **ds***).

The MASTER shall set TRANSFER_COUNT_ZERO if ~PARTIAL & (FETCH_ADD_BIG | FETCH_ADD_LITTLE) & it has asserted **ds*** during the first data beat | (~PARTIAL & (MASK_SWAP | COMPARE_SWAP) | (PARTIAL & (FETCH_ADD_BIG | FETCH_ADD_LITTLE)) & it has released **ds*** during the second data beat) | PARTIAL & (MASK_SWAP | COMPARE_SWAP) & it has asserted **ds*** during the third data beat.

The master shall set TRANSFER_COUNT_ZERO if MULTIPLE_PACKET_MODE & END_OF_DATA_STATUS.

The MASTER may set TRANSFER_COUNT_ZERO if COMPELLED_STATUS & (END_OF_DATA_STATUS | BEAT_ERROR_STATUS). The MASTER shall ignore data received after END_OF_DATA_STATUS has indicated it is invalid.

If the MASTER does not have knowledge of the slave's algorithm for asserting **BE*** with respect to the beat in which the error occurred (i.e., the number of pipeline stages), it shall ignore all the data transferred during the transaction.

If the MASTER does have knowledge of the slave's **BE*** algorithm, it may use the data transferred before the error occurred.

CONNECTION_STATUS. Modules shall set CONNECTION_STATUS when MASTER & **~AIf** | PARTICIPATING & (**AS*** & **DS*** | **~AS***).

DISCONNECTION_END. Modules shall set DISCONNECTION_END when DISCONNECTION_PHASE & (**~AKf** | the next **AS***). Slaves shall guarantee that the delay through their **AS*** logic is greater than or equal to the delay through their **AK*** logic with respect to the requirement of this paragraph.

6.2.2 Signal definition

6.2.2.1 Synchronization signals

6.2.2.1.1 AS* (address sync)

The MASTER shall assert **as*** when ~INIT & MASTER & TRANSACTION_BEGIN & TRANSACTION_COMPLETE & **~AKf** & **AI*** & ~BUS_HOLD & CONNECTION_PHASE and continue to assert it until INIT | ~MASTER | TRANSFER_COUNT_ZERO & **et*** & CONNECTION_INFO_CAPTURED.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2.2.1.2 AK* (address acknowledge)

All modules shall assert **ak*** when ~INIT & **AS*** and continue to assert it until INIT | ~**AS*** & DISC_INFO_CAPTURED & ~**dk*** & ~**di*** & ~**ds***.

6.2.2.1.3 AI* (address acknowledge inverse)

All modules shall assert **ai*** while ~(POWER_UP & ~BUS_IDLE_1US) & (INIT | the module detects the release of **AS***). Modules shall continue to assert **ai*** until **AS*** & ADDR_DECODED & TRANSACTION_COMPLETE.

6.2.2.1.4 DS* (data sync)

The MASTER shall assert **ds*** when ~INIT & MASTER & (~TRANSFER_COUNT_ZERO & (the rising edge of DATA_PHASE | the rising edge of DATA_INFO_CAPTURED) and continue to assert it until INIT | ~MASTER | the rising edge of DATA_INFO_CAPTURED & ~TRANSFER_COUNT_ZERO | ~**as** & (~BROADCAST_STATUS & ~**DK*** | BROADCAST_STATUS & ~**DKf**).

6.2.2.1.5 DK* (data acknowledge)

Modules shall assert **dk*** when ~INIT & **AS*** & **DS*** & PARTICIPATING & ~(SLAVE_WRITE | BROADCAST_STATUS).

If SLAVE_WRITE | BROADCAST_STATUS:

1) The module(s) sourcing the data shall assert **dk*** when ~INIT & **AS*** & **DS*** & PARTICIPATING and the data is valid.
2) Modules not sourcing the data shall assert **dk*** when ~INIT & **AS*** & **DS*** & **DK***.

Modules shall assert **dk*** until INIT | ~**DS*** & DATA_INFO_CAPTURED & **di*** | ~**AS***.

6.2.2.1.6 DI* (data acknowledge inverse)

Modules shall assert **di*** when ~INIT & **AS*** & ~**DS*** & PARTICIPATING & ~(SLAVE_WRITE | BROADCAST_STATUS).

If SLAVE_WRITE | BROADCAST_STATUS:

1) The module(s) sourcing the data shall assert **di*** when ~INIT & **AS*** & ~**DS*** & PARTICIPATING and the data is valid.
2) Modules not sourcing the data shall assert **di*** when ~INIT & **AS*** & ~**DS*** & **DI*.**

Modules shall assert **di*** until INIT | MULTIPLE_PACKET_MODE & ~PARTICIPATING | **DS*** & DATA_INFO_CAPTURED & **dk*** | ~**AS***. In the CONNECTION_PHASE, if PARTICIPATING, **DI*** shall be asserted prior to the release of **ai***.

6.2.2.2 CM[7..0]* (command)

The signals **cm[7..0]*** and **cp*** shall be stable at the connection to the bus prior to the transition of **as*** or **ds*** marking their validity. If necessary module designers shall delay transitions of **as*** and **ds*** more than is required in 6.2.2.1.1 and 6.2.2.1.4 to meet this requirement.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.2.2.1 CM0* (command 0)

The MASTER shall assert **cm0*** when ~INIT & MASTER & (CONNECTION_PHASE & TRANS_CMD_0 | DATA_PHASE & (LOCK_CMD_0 | ~COMPELLED_STATUS & PACKET_REQUEST) | DISCONNECTION_PHASE & LOCK_HOLD).

6.2.2.2.2 CM1* (command 1)

The MASTER shall assert **cm1*** when ~INIT & MASTER & (CONNECTION_PHASE & TRANS_CMD_1 | DATA_PHASE & (LOCK_CMD_1 | TRANSACTION_FLAG_STATUS) | DISCONNECTION_PHASE & MORE & (SPLIT_STATUS & READ_UNLOCKED | WRITE_UNLOCKED)).

6.2.2.2.3 CM2* (command 2)

The MASTER shall assert **cm2*** when ~INIT & MASTER & (CONNECTION_PHASE & TRANS_CMD_2 | DATA_PHASE & LOCK_CMD_2).

6.2.2.2.4 CM3* (command 3)

The MASTER shall assert **cm3*** when ~INIT & MASTER & CONNECTION_PHASE & TRANS_CMD_3.

6.2.2.2.5 CM4* (command 4)

The MASTER shall assert **cm4*** when ~INIT & MASTER & (CONNECTION_PHASE & WRITE_CMD | (DATA_PHASE | DISCONNECTION_PHASE) & INTERVENTION_STATUS).

6.2.2.2.6 CM5* (command 5)

The MASTER shall assert **cm5*** when ~INIT & MASTER & (CONNECTION_PHASE & DATA_WIDTH_0 | DATA_PHASE & DATA_LENGTH_0 | DISCONNECTION_PHASE & DATA_LENGTH_0).

6.2.2.2.7 CM6* (command 6)

The MASTER shall assert **cm6*** when ~INIT & MASTER & (CONNECTION_PHASE & DATA_WIDTH_1 | DATA_PHASE & DATA_LENGTH_1 | DISCONNECTION_PHASE & DATA_LENGTH_1).

6.2.2.2.8 CM7* (command 7)

The MASTER shall assert **cm7*** when ~INIT & MASTER & (CONNECTION_PHASE & ADDR_WIDTH | DATA_PHASE & DATA_LENGTH_2 | DISCONNECTION_PHASE & DATA_LENGTH_2).

6.2.2.2.9 CP* (command parity)

The MASTER shall assert **cp*** if it is asserting an even number of **cm[ ]*** signals.

6.2.2.3 CA[2..0]* (capability)

The signals **ca[2..0]*** shall be stable at the connection to the bus prior to the release of **ai*** which marks their validity. If necessary module designers shall delay the release of **ai*** more than is required in 6.2.2.1.3 to meet this requirement.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2.2.3.1 CA0* (TS*)

Modules shall assert **ca0*** when LOW_SPEED & CONNECTION_PHASE and shall continue to assert **ca0*** until INIT | DISCONNECTION_END.

6.2.2.3.2 CA1* (CO*)

Modules shall assert **ca1*** when COMPELLED and shall continue to assert **ca1*** until INIT | DISCONNECTION_END.

6.2.2.3.3 CA2* (SR*)

Modules shall assert **ca2*** when SPLIT and shall continue to assert **ca2*** until INIT | DISCONNECTION_END.

6.2.2.4 ST[7..0]* (status)

6.2.2.4.1 ST0* (TE*)

Modules shall assert **st0*** when ~INIT & TRANSACTION_ERROR & (**AS*** & ~**AIf** |~**AS*** & **ak***) and shall continue to assert it until INIT | CONNECTION_STATUS | ~PARTICIPATING & (~**AIf** & **AS*** | the next ~**AS***).

The signal **st0*** shall be stable at the connection to the bus prior to the release of **ak*** which marks its validity. If necessary module designers shall delay the release of **ak*** more than is required in 6.2.2.1.2 to meet this requirement.

6.2.2.4.2 ST1* (BS*/ED*)

Modules shall assert **st1*** when BUSY and shall continue to assert **st1*** until INIT | DISCONNECTION_END.

Modules shall assert **st1*** when END_OF_DATA | PACKET_NAK, and shall continue to assert **st1*** until INIT | DISCONNECTION_END.

The signal **st1*** shall be stable at the connection to the bus prior to the release of **ai***, **dk***, or **di*** which marks its validity. If necessary module designers shall delay the release of **ai***, **dk***, or **di*** more than is required in 6.2.2.1.3, 6.2.2.1.5, or 6.2.2.1.6 to meet this requirement.

6.2.2.4.3 ST2* (SL*)

Modules shall assert **st2*** when SELECTED and shall continue to assert **st2*** until INIT | ~SELECTED | DISCONNECTION_END.

The signal **st2*** shall be stable at the connection to the bus prior to the release of **ai***, **dk***, or **di*** which marks its validity. If necessary module designers shall delay the release of **ai***, **dk***, or **di*** more than is required in 6.2.2.1.3, 6.2.2.1.5, and 6.2.2.1.6 to meet this requirement.

6.2.2.4.4 ST3* (BC*)

Modules shall assert **st3*** when BROADCAST | BROADCALL_ADDRESS | SNARF_DATA | MULTIPLE_PACKET_MODE & CACHE & CACHE_COMMAND and shall continue to assert **st3*** until INIT | DISCONNECTION_END.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The signal **st3*** shall be stable at the connection to the bus prior to the release of **ai*** which marks its validity. If necessary module designers shall delay the release of **ai*** more than is required in 6.2.2.1.3 to meet this requirement.

6.2.2.4.5 ST4* (TF*)

Modules shall assert **st4*** when CACHE_CMD & ASSERT_CS | MAILBOX_SELECTED & MESSAGE_CONFLICT and shall continue to assert **st4*** until INIT | ~ASSERT_CS & CACHE_CMD | ~MESSAGE_CONFLICT & MAILBOX_SELECTED | DISCONNECTION_END.

The signal **st4*** shall be stable at the connection to the bus prior to the release of **ai***, **dk***, or **di*** which marks its validity. If necessary module designers shall delay the release of **ai***, **dk***, or **di*** more than is required in 6.2.2.1.3, 6.2.2.1.5, and 6.2.2.1.6 to meet this requirement.

6.2.2.4.6 ST5* (IV*)

Modules shall assert **st5*** when INTERVENTION and shall continue to assert **st5*** until INIT | ~INTERVENTION | DISCONNECTION_END.

The signal **st5*** shall be stable at the connection to the bus prior to the release of **ai***, **dk***, or **di*** which marks its validity. If necessary module designers shall delay the release of **ai***, **dk***, or **di*** more than is required in the equation in 6.2.2.1.3, 6.2.2.1.5, and 6.2.2.1.6 to meet this requirement.

6.2.2.4.7 ST6* (BE*)

Modules shall assert **st6*** when BEAT_ERROR and shall continue to assert **st6*** until INIT | DISCONNECTION_END.

The signal **st6*** shall be stable at the connection to the bus prior to the release of **ai***, **dk***, or **di*** which marks its validity. If necessary module designers shall delay the release of **ai***, **dk***, or **di*** more than is required in 6.2.2.1.3, 6.2.2.1.5, or 6.2.2.1.6 to meet this requirement.

6.2.2.4.8 ST7* (WT*)

Modules shall assert **st7*** when WAIT and shall continue to assert **st7*** until INIT | DISCONNECTION_END.

The signal **st7*** shall be stable at the connection to the bus prior to the release of **ai***, **dk***, or **di*** which marks its validity. If necessary module designers shall delay the release of **ai***, **dk***, or **di*** more than is required in 6.2.2.1.3, 6.2.2.1.5, or 6.2.2.1.6 to meet this requirement.

6.2.2.5 Information fields

6.2.2.5.1 AD[63..0]* (address/data)

Modules shall release **ad[ ]*** if INIT.

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ~ADDR_WIDTH & ~DATA_WIDTH_1 & ~DATA_WIDTH_0 the MASTER shall activate **ad[31..2]*** with the address, release **ad[63..32]***, and may activate **ad[1..0]*** with user defined data.

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ~ADDR_WIDTH & ~DATA_WIDTH_1 & DATA_WIDTH_0 the MASTER shall activate **ad[31..3]*** with the address, release **ad[63..32]***, and may activate **ad[2..0]*** with user defined data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ~ADDR_WIDTH & DATA_WIDTH_1 & ~DATA_WIDTH_0 the MASTER shall activate **ad[31..4]*** with the address, release **ad[63..32]***, and may activate **ad[3..0]*** with user defined data.

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ~ADDR_WIDTH & DATA_WIDTH_1 & DATA_WIDTH_0 the MASTER shall activate **ad[31..5]*** with the address, release **ad[63..32]***, and may activate **ad[4..0]*** with user defined data.

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ADDR_WIDTH & ~DATA_WIDTH_1 & ~DATA_WIDTH_0 the MASTER shall activate **ad[63..2]*** with the address and may activate **ad[1..0]*** with user defined data.

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ADDR_WIDTH & ~DATA_WIDTH_1 & DATA_WIDTH_0 the MASTER shall activate **ad[63..3]*** with the address and may activate **ad[2..0]*** with user defined data.

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ADDR_WIDTH & DATA_WIDTH_1 & ~DATA_WIDTH_0 the MASTER shall activate **ad[63..4]*** with the address and may activate **ad[3..0]*** with user defined data.

When ~INIT & CONNECTION_PHASE & ~(READ_RESPONSE | WRITE_RESPONSE) & ADDR_WIDTH & DATA_WIDTH_1 & DATA_WIDTH_0 the MASTER shall activate **ad[63..5]*** with the address and may activate **ad[4..0]*** with user defined data.

When ~INIT & CONNECTION_PHASE & (READ_RESPONSE | WRITE_RESPONSE) the MASTER shall

1) assert **ad[31..28]***,
2) activate **ad[27..12]*** with the requestor's global identification,
3) activate **ad[5..0]*** with a transaction identification,

and release **ad[63..32]*** and **ad[11..6]***.

When ~INIT & DISCONNECTION_PHASE & (READ_RESPONSE | WRITE_RESPONSE) the MASTER shall

1) activate **ad[15..8]*** with the requestor's original transaction priority,
2) activate **ad2*** with the selected status which was signaled in the request transaction,
3) activate **ad1*** with the conflict status which was signaled in the request transaction,
4) activate **ad0*** with the transaction error status which was signaled during the request transaction,
5) and release **ad[63..32]***, **ad[31..16]***, and **ad[7..3]***.

When ~INIT & DISCONNECTION_PHASE & (SHARED_RESPONSE | MODIFIED_RESPONSE) the MASTER shall

1) assert **ad[31..28]***,
2) activate **ad[27..12]*** with the requestor's global identification,
3) activate **ad[11..4]*** with the requestor's original transaction priority,
4) activate **ad2*** with the selected status which was signaled in the request transaction,
5) activate **ad1*** with the conflict status which was signaled in the request transaction,
6) activate **ad0*** with the transaction error status which was signaled during the request transaction,

and release **ad[63..32]*** and **ad3***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

When ~INIT & DISCONNECTION_PHASE & (SPLIT_STATUS | WRITE_UNLOCKED | ADDRESS_ONLY_UNLOCKED | WRITE_LOCKED | ADDRESS_ONLY_LOCKED | WRITE_PARTIAL | WRITE_PARTIAL_LOCKED | WRITE_NO_ACKNOWLEDGE | WRITE_INVALID | COPY_BACK | INVALIDATE) the MASTER shall

1) activate **ad[31..16]*** with the requestor's global identification,
2) activate **ad[15..8]*** with the requestor's original transaction priority,
3) activate **ad[5..0]*** with a transaction identification,

and release **ad[63..32]*** and **ad[7..6]***.

When ~INIT & DATA_PHASE & WRITE_CMD the MASTER shall activate **ad[31..0]*** with the write data associated with the current beat or packet.

When ~INIT & DATA_PHASE & WRITE_CMD & (DATA_WIDTH_1 | DATA_WIDTH_0) the MASTER shall activate **ad[63..32]*** with the write data associated with the current beat or packet.

When a PARTICIPATING slave is responsible for transmitting data to the MASTER it shall activate **ad[31..0]*** with the write data associated with the current beat or packet in response to transitions of **DS***.

When a PARTICIPATING slave is responsible for transmitting data to the MASTER and (64_BIT_DATA; | 128_BIT_DATA | 256_BIT_DATA), it shall activate **ad[63..32]*** with the write data associated with the current beat or packet in response to transitions of **DS***.

If a transaction is selected which uses the packet protocols, **ad[ ]*** shall comply with the packet protocol as specified in 6.2.3.21. Packets shall always be address aligned on a packet size modulus boundary.

The **ad[ ]*** signals shall be stable at the connection to the bus prior to the transition of **as*** marking their validity. If necessary module designers shall delay transitions of **as*** more than is required in 6.2.2.1.1 to meet this requirement.

In compelled mode the **ad[ ]*** signals shall be stable at the connection to the bus prior to the transition of **ds*** marking their validity if the MASTER is transmitting. If necessary module designers shall delay transitions of **ds*** more than is required in 6.2.2.1.4 to meet this requirement.

In compelled mode the **ad[ ]*** signals shall be stable at the connection to the bus prior to the transition of **dk*** or **di*** which marks their validity if one or more slaves are transmitting. If necessary, module designers shall delay the transition of **dk*** or **di*** more than is required in 6.2.2.1.5 or 6.2.2.1.6 to meet this requirement.

In compelled data beats where SLAVE_WRITE | BROADCAST_STATUS the **ad[ ]*** signals shall be stable at the connection to the bus prior to the assertion of **dk*** or **di*** which mark their validity. If necessary module designers shall delay the assertion of **dk*** or **di*** more than is required in 6.2.2.1.5 or 6.2.2.1.6 to meet this requirement.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The lane deselect assignments shall be as follows:

ad[31..0]*	Byte lane	Lane deselect
ad31*	D[255..248]*	L31*
ad30*	D[247..240]*	L30*
ad29*	D[239..232]*	L29*
ad28*	D[231..224]*	L28*
ad27*	D[223..216]*	L27*
ad26*	D[215..208]*	L26*
ad25*	D[207..200]*	L25*
ad24*	D[199..192]*	L24*
ad23*	D[191..184]*	L23*
ad22*	D[183..176]*	L22*
ad21*	D[175..168]*	L21*
ad20*	D[167..160]*	L20*
ad19*	D[159..152]*	L19*
ad18*	D[151..144]*	L18*
ad17*	D[143..136]*	L17*
ad16*	D[135..128]*	L16*
ad15*	D[127..120]*	L15*
ad14*	D[119..112]*	L14*
ad13*	D[111..104]*	L13*
ad12*	D[103..96]*	L12*
ad11*	D[95..88]*	L11*
ad10*	D[87..80]*	L10*
ad9*	D[79..72]*	L9*
ad8*	D[71..64]*	L8*
ad7*	AD[63..56]*	L7*
ad6*	AD[55..48]*	L6*
ad5*	AD[47..40]*	L5*
ad4*	AD[39..32]*	L4*
ad3*	AD[31..24]*	L3*
ad2*	AD[23..16]*	L2*
ad1*	AD[15..8]*	L1*
ad0*	AD[7..0]*	L0*

During the first beat of a transaction when ~INIT & DATA_PHASE & partial, the MASTER shall activate **ad[31..0]*** with the lane disable information as specified in the above table, and it shall release **ad[63..32]***. Modules should release lane deselect signals if the lanes they control were not selected by DATA_WIDTH_1, DATA_WIDTH_0 in the connection phase.

6.2.2.5.2 D[255.64]* (data)

When ~INIT & DATA_PHASE & WRITE_CMD & DATA_WIDTH_1 the MASTER shall activate **d[127..64]*** with the write data associated with the current beat or packet.

When ~INIT & DATA_PHASE & WRITE_CMD & (DATA_WIDTH_1 & DATA_WIDTH_0) the MASTER shall activate **d[255..128]*** with the write data associated with the current beat or packet.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

When a PARTICIPATING slave is responsible for transmitting data to the MASTER and (128_BIT_DATA | 256_BIT_DATA), it shall activate **d[127..64]*** with the write data associated with the current beat or packet in response to transitions of **DS***.

When a PARTICIPATING slave is responsible for transmitting data to the MASTER and 256_BIT_DATA, it shall activate **d[255..128]*** with the write data associated with the current beat or packet in response to transitions of **DS***.

During the first beat of a transaction when ~INIT & DATA_PHASE & partial, the MASTER shall release **d[255..64]***.

If a transaction is selected which uses the packet protocols, **d[ ]*** shall comply with the packet protocol as specified in 6.2.3.21.

In compelled mode the **d[ ]*** signals shall be stable at the connection to the bus prior to the transition of **ds*** marking their validity if the MASTER is transmitting. If necessary module designers shall delay transitions of **ds*** more than is required in 6.2.2.1.4 to meet this requirement.

In compelled mode the **d[ ]*** signals shall be stable at the connection to the bus prior to the transition of **dk*** or **di*** which marks their validity if one or more slaves are transmitting. If necessary module designers shall delay the transition of **dk*** or **di*** more than is required in 6.2.2.1.5 or 6.2.2.1.6 to meet this requirement.

In compelled data beats where SLAVE_WRITE | BROADCAST_STATUS the **ad[ ]*** signals shall be stable at the connection to the bus prior to the assertion of **dk*** or **di*** which mark their validity. If necessary module designers shall delay the assertion of **dk*** or **di*** more than is required in 6.2.2.1.5 or 6.2.2.1.6 to meet this requirement.

6.2.2.5.3 BP[31..0]* (bus parity)

Modules shall release **bp[ ]*** if INIT.

The parity bit assignments shall be as follows:

Byte lane	Parity signal
D[255..248]*	BP31*
D[247..240]*	BP30*
D[239..232]*	BP29*
D[231..224]*	BP28*
D[223..216]*	BP27*
D[215..208]*	BP26*
D[207..200]*	BP25*
D[199..192]*	BP24*
D[191..184]*	BP23*
D[183..176]*	BP22*
D[175..168]*	BP21*
D[167..160]*	BP20*
D[159..152]*	BP19*
D[151..144]*	BP18*
D[143..136]*	BP17*
D[135..128]*	BP16*
D[127..120]*	BP15*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Byte lane	Parity signal
D[119..112]*	BP14*
D[111..104]*	BP13*
D[103..96]*	BP12*
D[95..88]*	BP11*
D[87..80]*	BP10*
D[79..72]*	BP9*
D[71..64]*	BP8*
AD[63..56]*	BP7*
AD[55..48]*	BP6*
AD[47..40]*	BP5*
AD[39..32]*	BP4*
AD[31..24]*	BP3*
AD[23..16]*	BP2*
AD[15..8]*	BP1*
AD[7..0]*	BP0*

If a module asserts an even number of bits in a byte lane specified in the left column, it shall assert the parity bit specified in the right column.

When ~INIT & CONNECTION_PHASE & (~ADDR_WIDTH | READ_RESPONSE | WRITE_RESPONSE) the MASTER shall activate **bp[3..0]*** with correct parity for **AD[31..0]***.

When ~INIT & CONNECTION_PHASE & ADDR_WIDTH & ~(READ_RESPONSE | WRITE_RESPONSE) the MASTER shall activate **bp[7..0]*** with correct parity for **AD[63..0]***.

When ~INIT & DATA_PHASE & WRITE_CMD, the MASTER shall activate **bp[3..0]*** with correct parity for **AD[31..0]***.

When ~INIT & DATA_PHASE & WRITE_CMD & (DATA_WIDTH_1 | DATA_WIDTH_0), the MASTER shall activate **bp[7..4]*** with correct parity for **AD[63..32]***.

When ~INIT & DATA_PHASE & WRITE_CMD & DATA_WIDTH_1, the MASTER shall activate **bp[15..8]*** with correct parity for **D[127..64]***.

When ~INIT & DATA_PHASE & WRITE_CMD & DATA_WIDTH_1 & DATA_WIDTH_0, the MASTER shall activate **bp[31..16]*** with correct parity for **D[255..128]***.

When ~INIT & DATA_PHASE and a PARTICIPATING slave is responsible for transmitting data to the MASTER it shall activate **bp[3..0]*** with correct parity for **AD[31..0]***.

When ~INIT & DATA_PHASE and a PARTICIPATING slave is responsible for transmitting data to the MASTER and (64_BIT_DATA; | 128_BIT_DATA | 256_BIT_DATA), it shall activate **bp[7..4]*** with correct parity for **AD[63..32]***.

When ~INIT & DATA_PHASE and a PARTICIPATING slave is responsible for transmitting data to the MASTER and (128_BIT_DATA | 256_BIT_DATA), it shall activate **bp[15..8]*** with correct parity for **D[127..64]***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

When ~INIT & DATA_PHASE and a PARTICIPATING slave is responsible to for transmitting data to the MASTER and 256_BIT_DATA, it shall activate **bp[31..16]*** with correct parity for **D[255..128]***.

During the first beat of a transaction when ~INIT & DATA_PHASE & PARTIAL, the MASTER shall drive correct parity on all the lines associated with the byte lanes indicated by DATA_WIDTH_1, DATA_WIDTH_0 for that transaction. PARTICIPATING modules shall use DATA_WIDTH 1, DATA_WIDTH_0 to determine which byte lanes have activated parity.

When ~INIT & DISCONNECTION_PHASE & (WRITE_CMD | SPLIT_STATUS) the MASTER shall activate correct parity for **AD[31..0]*.**

If a transaction is selected which uses the packet protocols, **bp[ ]*** shall comply with the packet protocol as specified in 6.2.3.21.

The **bp[ ]*** signals shall be stable at the connection to the bus prior to the transition of **as*** marking their validity. If necessary module designers shall delay transitions of **as*** more than is required in 6.2.2.1.1 to meet this requirement.

In compelled mode the **bp[ ]*** signals shall be stable at the connection to the bus prior to the transition of **ds*** marking their validity if the MASTER is transmitting. If necessary module designers shall delay transitions of **ds*** more than is required in 6.2.2.1.4 to meet this requirement.

In compelled mode the **bp[ ]*** signals shall be stable at the connection to the bus prior to the transition of **dk*** or **di*** which marks their validity if one or more slaves are transmitting. If necessary module designers shall delay the transition of **dk*** or **di*** more than is required in 6.2.2.1.5 or 6.2.2.1.6 to meet this requirement.

In compelled data beats where SLAVE_WRITE | BROADCAST_STATUS the **bp[ ]*** signals shall be stable at the connection to the bus prior to the assertion of **dk*** or **di*** which mark their validity. If necessary module designers shall delay the assertion of **dk*** or **di*** more than is required in 6.2.2.1.5 or 6.2.2.1.6 to meet this requirement.

6.2.2.5.4 TG[7..0]* (tag)

The **tg[7..0]*** shall follow the same timing and other specifications as **ad[31..0]*** except that it is left to the system designer to determine when they shall carry information.

The number of **tg[ ]*** signals active shall be as specified in 7.2.4.

6.2.2.5.5 TP* (tag parity)

If TAG_ENABLE is set and a module is asserting an even number of **tg[ ]*** it shall assert **tp***.

6.2.2.5.6 ET*

Modules shall release **et*** if INIT.

The MASTER shall assert **et*** when it detects **AIf** at logic zero and prior to the release of **as*** during its first transaction of its current tenure. The MASTER shall maintain **et*** asserted until **~rq1*** & **~rq0*** & **~AKf** and it has released **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, **tp***, **cm[ ]***, and **cp***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2.3 Protocol definition

6.2.3.1 Master connection phase

1) The MASTER shall set the attributes specified in 6.2.1.1, 6.2.1.2, and 6.2.1.4.
2) The MASTER shall clear TRANSACTION_TIMEOUT.
3) The MASTER shall wait until ~**AKf** & **AI*** | TRANSACTION_TIMEOUT.
4) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
5) The MASTER shall activate **ad[ ]***, **bp[ ]***, **tg[ ]***, **tp***, **cm[ ]***, **cp***, and **ca[ ]*** with the appropriate information.
6) The MASTER shall set TRANSACTION_BEGIN.
7) The MASTER shall assert **as*** and **ak***.
8) The MASTER shall release **ai***.
9) The MASTER shall wait until ~**AIf** | TRANSACTION_TIMEOUT and evaluate the information on **ST[ ]*** and **CA[ ]*.**
10) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
11) The MASTER shall release **te*** and assert **et***.
12) The MASTER shall set the attributes as specified in 6.2.1.7.1.
13) If TRANSFER_COUNT_ZERO the MASTER shall proceed to the disconnection phase specified in 6.2.3.22.
14) If COMPELLED_STATUS & (READ_UNLOCKED | WRITE_UNLOCKED | READ_LOCKED | WRITE_LOCKED | READ_PARTIAL | WRITE_PARTIAL | READ_PARTIAL_LOCKED | WRITE_PARTIAL_LOCKED | READ_RESPONSE | SHARED_RESPONSE | MODIFIED_RESPONSE | WRITE_NO_ACKNOWLEDGE | READ_INVALID | WRITE_INVALID | READ_SHARED | COPY_BACK | READ_MODIFIED) the MASTER may proceed to the odd beat of the compelled data phase as specified in 6.2.3.3.
15) If ~MULTIPLE_PACKET_MODE & (READ_UNLOCKED | WRITE_UNLOCKED | READ_RESPONSE | WRITE_NO_ACKNOWLEDGE | READ_INVALID | WRITE_INVALID | READ_SHARED | COPY_BACK | SHARED_RESPONSE | MODIFIED_RESPONSE | READ_MODIFIED) the MASTER may proceed to the single packet data phase as specified in 6.2.3.13.
16) If MULTIPLE_PACKET_MODE & (READ_UNLOCKED | WRITE_UNLOCKED | READ_RESPONSE | WRITE_NO_ACKNOWLEDGE | READ_INVALID | WRITE_INVALID | READ_SHARED | COPY_BACK | SHARED_RESPONSE | MODIFIED_RESPONSE | READ_MODIFIED) the MASTER may proceed to the multiple packet data phase as specified in 6.2.3.15.
17) The MASTER may proceed to the disconnection phase as specified in 6.2.3.22.

6.2.3.2 Slave connection phase

A slave shall wait until it detects the assertion of **AS*** and then it shall

1) Assert **ak*** if ~READY_FOR_WITHDRAWAL.
2) Evaluate **AD[ ]***, **BP[ ]***, **TG[ ]***, **TP***, **CM[ ]*** and **CP***.
3) Wait until ADDR_DECODED.
4) Activate **st[7..1]*** and **ca[2..0]*** with the appropriate information.
5) If locked set the RESOURCE_LOCK associated with the address.
6) Assert **di*** if PARTICIPATING.
7) Then release **ai***.
8) Then wait until CONNECTION_STATUS | ~PARTICIPATING & (~**AIf** & **AS***| the next ~**AS***).
9) Then release **TE*.**
10) Evaluate the information on **CA[ ]***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11) If ~PARTICIPATING wait until DISCONNECTION_PHASE and proceed to the disconnection phase as specified in 6.2.3.23.
12) Wait until DATA_PHASE | DISCONNECTION_PHASE.
13) If DATA_PHASE & COMPELLED_STATUS proceed to the odd beat specified in 6.2.3.4.
14) If DATA_PHASE & ~COMPELLED_STATUS & ~MULTIPLE_PACKET_MODE proceed to the odd packet specified in 6.2.3.14.
15) If DATA_PHASE & MULTIPLE_PACKET_MODE, proceed to the odd packet as specified in 6.2.3.16.
16) Proceed to the disconnection phase as specified in 6.2.3.23.

6.2.3.3 Master compelled data phase—first odd beat

1) The MASTER shall activate **cm[ ]*** and **cp*** with the appropriate command.
2) If PARTIAL the MASTER shall activate **ad[ ]*** with the appropriate lane deselect information and then proceed to 7).
3) If MASK_SWAP the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate mask operand and proceed to 7).
4) If COMPARE_SWAP the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate compare operand and proceed to 7).
5) If (FETCH_ADD_BIG | FETCH_ADD_LITTLE) the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate add operand and proceed to 7).
6) If WRITE_CMD the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate data and proceed to 7). If ~WRITE_CMD proceed to 8).
7) The MASTER shall activate **bp[ ]***, **tg[ ]***, and **tp*** with appropriate information and shall then proceed to 9).
8) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp***.
9) The MASTER shall then assert **ds***.
10) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
11) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
12) If ~PARTIAL & ~WRITE_CMD the MASTER shall evaluate the information on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP***.
13) If ~PARTIAL & (FETCH_ADD_BIG | FETCH_ADD_LITTLE) | SPLIT_STATUS & PARTIAL & ~WRITE_CMD the MASTER shall proceed to the disconnection phase as specified in 6.2.3.22.
14) The MASTER shall proceed to an even beat as specified in 6.2.3.5 if ~TRANSFER_COUNT_ZERO or to the disconnection phase as specified in 6.2.3.22 if TRANSFER_COUNT_ZERO.

6.2.3.4 Slave compelled data phase—first odd beat

A slave shall

1) If ~WRITE_STATUS & (SELECTED & SLAVE_WRITE | BROADCAST | SNARF_DATA) wait until **DK*** is asserted, assert **dk***, evaluate the data on **AD[ ]***, **D[ ]*** and proceed to 10).
2) If ~WRITE_STATUS & (INTERVENTION | SELECTED & BROADCAST_STATUS) activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information, assert **dk*** and proceed to 11).
3) Assert **dk***.
4) If ~WRITE_STATUS & SELECTED activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and proceed to 11).
5) If PARTIAL & SELECTED evaluate the lane deselect information on **AD[31..0]*** and proceed to 10).
6) If MASK_SWAP & SELECTED evaluate the mask operand on **AD[ ]*** and **D[ ]*** and proceed to 10).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7) If COMPARE_SWAP & SELECTED evaluate the compare operand on **AD[ ]*** and **D[ ]*** and proceed to 10).
8) If (FETCH_ADD_BIG | FETCH_ADD_LITTLE) & SELECTED evaluate the add operand on **AD[ ]*** and **D[ ]*** and proceed to 10).
9) If WRITE_STATUS & (SELECTED | BROADCAST | SNARF_DATA) evaluate the data on **AD[ ]*** and **D[ ]***.
10) Evaluate the information on **BP***, **TG[ ]***, and **TP***.
11) Activate **st6***, **st1***, and **st0*** with the appropriate status.
12) Then release **di***.
13) Wait until **~AS*** | **~DS***.
14) If DISCONNECTION_PHASE release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and proceed to the disconnection phase as specified in 6.2.3.23.
15) If DATA_PHASE proceed to an even beat as specified in 6.2.3.6.

6.2.3.5 Master compelled data phase—first even beat

1) If PARTIAL & MASK_SWAP the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate mask operand and proceed to 7).
2) If PARTIAL & COMPARE_SWAP the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate compare operand and proceed to 7).
3) If PARTIAL & (FETCH_ADD_BIG | FETCH_ADD_LITTLE) the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate add operand and proceed to 7).
4) If COMPARE_SWAP | MASK_SWAP the MASTER shall activate **ad[ ]*** and **d[ ]***with the appropriate swap operand and proceed to 7).
5) If WRITE_CMD the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate data and proceed to 7).
6) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and proceed to 8).
7) The MASTER shall activate **bp[ ]***, **tg[ ]***, and **tp*** with appropriate information.
8) The MASTER shall then release **ds***.
9) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
10) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
11) If ~WRITE_CMD the MASTER shall evaluate the information on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP***.
12) If PARTIAL & (FETCH_ADD_BIG | FETCH_ADD_LITTLE) | ~PARTIAL & (MASK_SWAP | COMPARE_SWAP) the MASTER shall proceed to the disconnection phase as specified in 6.2.3.22.
13) The MASTER shall proceed to an odd beat as specified in 6.2.3.7 if ~TRANSFER_COUNT_ZERO or to the disconnection phase as specified in 6.2.3.22 if TRANSFER_COUNT_ZERO.

6.2.3.6 Slave compelled data phase—first even beat

A slave shall

1) If ~WRITE_STATUS & (SELECTED & SLAVE_WRITE | BROADCAST | SNARF_DATA) wait until **DI*** is asserted, assert **di***, evaluate the data on **AD[ ]***, **D[ ]*** and proceed to 10).
2) If ~WRITE_STATUS & (INTERVENTION | SELECTED & BROADCAST_STATUS) activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information, assert **di*** and proceed to 11).
3) Assert **di***.
4) If ~WRITE_STATUS & SELECTED activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and proceed to 11).
5) If PARTIAL & MASK_SWAP & SELECTED evaluate the mask operand on **AD[ ]*** and **D[ ]*** and proceed to 10).
6) If PARTIAL & COMPARE_SWAP & SELECTED evaluate the compare operand on **AD[ ]*** and **D[ ]*** and proceed to 10).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7) If PARTIAL & (FETCH_ADD_BIG | FETCH_ADD_LITTLE) & SELECTED evaluate the add operand on **AD[ ]*** and **D[ ]*** and proceed to 10).
8) If (MASK_SWAP | COMPARE_SWAP) & SELECTED evaluate the swap operand on **AD[ ]*** and **D[ ]*** and proceed to 10).
9) If WRITE_STATUS & (SELECTED | BROADCAST | SNARF_DATA) evaluate the data on **AD[ ]*** and **D[ ]***.
10) Evaluate the information on **BP***, **TG[ ]***, and **TP***.
11) Activate **st6***, **st1***, and **st0*** with the appropriate status.
12) Then release **dk***.
13) Wait until ~**AS*** | **DS***.
14) If DISCONNECTION_PHASE release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and proceed to the disconnection phase as specified in 6.2.3.23.
15) If DATA_PHASE proceed to an odd beat as specified in 6.2.3.8.

6.2.3.7 Master compelled data phase—second odd beat

1) If PARTIAL & (MASK_SWAP | COMPARE_SWAP) the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate swap operand and proceed to 4).
2) If WRITE_CMD the MASTER shall activate **ad[ ]*** and **d[ ]*** with the appropriate data and proceed to 7).
3) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and proceed to 5).
4) The MASTER shall activate **bp[ ]***, **tg[ ]***, and **tp*** with appropriate information.
5) The MASTER shall then assert **ds***.
6) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
7) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
8) If ~WRITE_CMD the MASTER shall evaluate the information on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP***.
9) If PARTIAL & (MASK_SWAP | COMPARE_SWAP) the MASTER shall proceed to the disconnection phase as specified in 6.2.3.22.
10) The MASTER shall proceed to an even beat as specified in 6.2.3.11 if ~TRANSFER_COUNT_ZERO or to the disconnection phase as specified in 6.2.3.22 if TRANSFER_COUNT_ZERO.

6.2.3.8 Slave compelled data phase—second odd beat

A slave shall

1) If ~WRITE_STATUS & (SELECTED & SLAVE_WRITE | BROADCAST | SNARF_DATA) wait until **DK*** is asserted and then evaluate the data on **AD[ ]*** and **D[ ]*** and proceed to 7).
2) If ~WRITE_STATUS & (INTERVENTION | SELECTED & BROADCAST_STATUS) activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and then assert **dk*** and proceed to 8).
3) Assert **dk***.
4) If ~WRITE_STATUS & SELECTED activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and proceed to 8).
5) If PARTIAL & (MASK_SWAP | COMPARE_SWAP) & SELECTED evaluate the swap operand on **AD[ ]*** and **D[ ]*** and proceed to 7).
6) If WRITE_STATUS & (SELECTED | BROADCAST | SNARF_DATA) evaluate the data on **AD[ ]*** and **D[ ]***.
7) Evaluate the information on **BP[ ]***, **TG[ ]***, and **TP***.
8) Activate **st6***, **st1***, and **st0*** with the appropriate status.
9) Then release **di***.
10) Wait until ~**AS*** | ~**DS***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11) If DISCONNECTION_PHASE release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and proceed to the disconnection phase as specified in 6.2.3.23.
12) If DATA_PHASE proceed to an even beat as specified in 6.2.3.12.

6.2.3.9 Master compelled data phase—third and subsequent odd beats

1) If WRITE_CMD the MASTER shall activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with appropriate information and shall then proceed to 3).
2) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp***.
3) The MASTER shall then assert **ds***.
4) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
5) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
6) If ~WRITE_CMD the MASTER shall evaluate the information on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP***.
7) The MASTER shall proceed to an even beat as specified in 6.2.3.11 if ~TRANSFER_COUNT_ZERO or to the disconnection phase as specified in 6.2.3.22 if TRANSFER_COUNT_ZERO.

6.2.3.10 Slave compelled data phase—third and subsequent odd beats

A slave shall

1) If ~WRITE_STATUS & (SELECTED & SLAVE_WRITE | BROADCAST | SNARF_DATA) wait until **DK*** is asserted and then evaluate the data on **AD[ ]*** and **D[ ]*** and proceed to 6).
2) If ~WRITE_STATUS & (INTERVENTION | SELECTED & BROADCAST_STATUS) activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and then assert **dk*** and proceed to 7).
3) Assert **dk***.
4) If ~WRITE_STATUS & SELECTED activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and proceed to 7).
5) If WRITE_STATUS & (SELECTED | BROADCAST | SNARF_DATA) evaluate the data on **AD[ ]*** and **D[ ]***.
6) Evaluate the information on **BP[ ]***, **TG[ ]***, and **TP***.
7) Activate **st6***, **st1***, and **st0*** with the appropriate status.
8) Then release **di***.
9) Wait until **~AS*** | **~DS***.
10) If DISCONNECTION_PHASE release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and proceed to the disconnection phase as specified in 6.2.3.23.
11) If DATA_PHASE proceed to an even beat as specified in 6.2.3.12.

6.2.3.11 Master compelled data phase—second and subsequent even beats

1) If WRITE_CMD the MASTER shall activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with appropriate information and shall then proceed to 3).
2) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp***.
3) The MASTER shall then release **ds***.
4) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
5) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
6) If ~WRITE_CMD the MASTER shall evaluate the information on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP***.
7) The MASTER shall proceed to an odd beat as specified in 6.2.3.9 if ~TRANSFER_COUNT_ZERO or to the disconnection phase as specified in 6.2.3.22 if TRANSFER_COUNT_ZERO.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.3.12 Slave compelled data phase—second and subsequent even beats

A slave shall

1) If ~WRITE_STATUS & (SELECTED & SLAVE_WRITE | BROADCAST | SNARF_DATA) wait until **DI*** is asserted and then evaluate the data on **AD[ ]*** and **D[ ]*** and proceed to 6).
2) If ~WRITE_STATUS & (INTERVENTION | SELECTED & BROADCAST_STATUS) activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and then assert **di*** and proceed to 7).
3) Assert **di***.
4) If ~WRITE_STATUS & SELECTED activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate information and proceed to 7).
5) If WRITE_STATUS & (SELECTED | BROADCAST | SNARF_DATA) evaluate the data on **AD[ ]*** and **D[ ]***.
6) Evaluate the information on **BP[ ]***, **TG[ ]***, and **TP***.
7) Activate **st6***, **st1***, and **st0*** with the appropriate status.
8) Then release **dk***.
9) Wait until **~AS* | DS***.
10) If DISCONNECTION_PHASE release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and proceed to the disconnection phase as specified in 6.2.3.23.
11) If DATA_PHASE proceed to an odd beat as specified in 6.2.3.10.

6.2.3.13 Master packet data phase—single packet mode

1) The MASTER shall activate **cm[ ]*** and **cp*** with the appropriate information.
2) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp***.
3) The MASTER shall then assert **ds***.
4) If WRITE_CMD the MASTER shall wait a minimum of one bit period time and then activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with appropriate packet transmission and when the parity bit has been transmitted proceed to 8).
5) If ~WRITE_CMD the MASTER shall evaluate the packet transmission on **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP*** and when the parity bit has been received proceed to 6).
6) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
7) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
8) The MASTER shall proceed to the disconnection phase as specified in 6.2.3.22.

6.2.3.14 Slave packet data phase—single packet mode

A slave shall

1) Assert **dk***, activate **st1*** and **st0*** with the appropriate status, and then release **di***. The completion of 1) may be delayed, as long as it is completed prior to 7).
2) If WRITE_STATUS evaluate the packet transmission on **AD[ ]***, **D[ ]***, **BP***, **TG[ ]***, and **TP*** and proceed to 7).
3) If ~WRITE_STATUS & (SELECTED & SLAVE_WRITE | BROADCAST) evaluate the packet transmission on **AD[ ]***, **D[ ]***, **BP***, **TG[ ]***, and **TP*** and proceed to 7).
4) If SELECTED & ~WRITE_STATUS & ~SLAVE_WRITE activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate packet transmission and proceed to 6).
5) If ~WRITE_STATUS & INTERVENTION activate **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** with the appropriate packet transmission and proceed to 6).
6) Release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp***.
7) Then wait until **~AS*** and proceed to the disconnection phase as specified in 6.2.3.23.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2.3.15 Master packet data phase—multiple packet mode—odd packet queueing

1) The MASTER shall activate **cm[ ]*** and **cp*** with the appropriate information.
2) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and prepare its queue to receive a packet.
3) The MASTER shall then assert **ds***.
4) If WRITE_CMD the MASTER shall queue the appropriate packet transmission.
5) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
6) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
7) The MASTER shall proceed to queue an even packet as specified in 6.2.3.17 if ~TRANSFER_COUNT_ZERO or to the disconnection phase as specified in 6.2.3.22 if TRANSFER_COUNT_ZERO.

6.2.3.16 Slave packet data phase—multiple packet mode—odd packet queueing

A slave shall

1) If WRITE_STATUS prepare its queue to receive a packet.
2) If SELECTED & ~WRITE_STATUS & ~SLAVE_WRITE queue the appropriate packet transmission.
3) If ~WRITE_STATUS & INTERVENTION queue the appropriate packet transmission.
4) Assert **dk***, activate **st7***, **st5***, **st4***, **st2***, **st1*** and **st0*** with the appropriate status, and then release **di***.
5) Wait until **~AS*** | **~DS***.
6) If DISCONNECTION_PHASE wait until the parity bit of the last queued packet is received or transmitted, release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp***, and proceed to the disconnection phase as specified in 6.2.3.23.
7) If DATA_PHASE proceed to queue an even packet as specified in 6.2.3.18.

6.2.3.17 Master packet data phase—multiple packet mode—even packet queueing

1) The MASTER shall activate **cm[ ]*** and **cp*** with the appropriate information.
2) If ~WRITE_CMD the MASTER shall release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp*** and prepare its queue to receive a packet.
3) The MASTER shall then release **ds***.
4) If WRITE_CMD the MASTER shall queue the appropriate packet transmission.
5) The MASTER shall wait until DATA_INFO_CAPTURED | TRANSACTION_TIMEOUT.
6) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recovery as specified in 6.2.3.24.
7) The MASTER shall proceed to queue an odd packet as specified in 6.2.3.15 if ~TRANSFER_COUNT_ZERO or to the disconnection phase as specified in 6.2.3.22 if TRANSFER_COUNT_ZERO.

6.2.3.18 Slave packet data phase—multiple packet mode—even packet queueing

A slave shall

1) If WRITE_STATUS prepare its queue to receive a packet.
2) If SELECTED & ~WRITE_STATUS & ~SLAVE_WRITE queue the appropriate packet transmission.
3) If ~WRITE_STATUS & INTERVENTION queue the appropriate packet transmission.
4) Assert **di***, activate **st7***, **st5***, **st4***, **st2***, **st1*** and **st0*** with the appropriate status, and then release **dk***.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5) Wait until ~**AS*** | **DS***.
6) If DISCONNECTION_PHASE wait until the parity bit of the last queued packet is received or transmitted, release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, and **tp***, and proceed to the disconnection phase as specified in 6.2.3.23.
7) If DATA_PHASE proceed to queue an odd packet as specified in 6.2.3.16.

6.2.3.19 Master packet data phase—multiple packet mode—packet transmission

The MASTER shall track, e.g., count, every transfer so that it has knowledge of the beginning and end of every packet.

Prior to initiation of the transmission of the first packet the MASTER shall wait at least one bit period past its initial assertion of **ds***.

To initiate the transmission of any subsequent packet the MASTER shall

1) Wait until it has made one **ds*** signal level change after the **ds*** signal level change that requested the packet, and then mark the packet state as ready.
2) If there is no packet currently being transmitted and the new packet is the only one in the ready state, wait at least one bit period past its last **ds*** signal level change.
3) If there is a packet currently being transmitted or packets ahead of this one in the transmit queue, wait until the parity bit of all previously queued ready packets have been transmitted.

6.2.3.20 Slave packet data phase—multiple packet mode—packet transmission

All participating slaves shall track, e.g., count, every transfer so that every participant has knowledge of the beginning and end of every packet.

To initiate the transmission of any packet a slave shall

1) If there is no packet currently being transmitted and the new packet is the only one in the ready state on any module in the logical bus, the slave may begin transmitting the new packet immediately.
2) If there is a packet currently being transmitted or there are packets ahead of this one on this logical bus transmit queue, the slave shall wait until the parity bits of all previously queued ready packets have been transmitted. Then the slave may begin to transmit the new packet when at least the full bit period has elapsed past the slave's reception of the last parity bit.

Any participating module may acquire any packet it chooses without any interaction with the above protocol unless packets are part of coherent shared memory. If packets are part of coherent shared memory, modules shall activate the appropriate status information during the acknowledge following the packet request.

6.2.3.21 Packet data protocol

Each local reference clock shall have a frequency tolerance of ±0.01% of nominal or better over all operating conditions. A module reports its packet speed as specified in 7.2.4.14.

Data is encoded using NRZ-I where a logical one is represented by a low asserted level during a bit period and a zero is represented by a high released level.

Serial skew (or jitter) shall be less than 15% of a clock period. This deviation of any edge from its nominal time shall be with respect to the sync edge at the beginning of the packet.

A packet shall not begin until **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP*** are released for at least 1 bit period. To begin transmission a sync word is sent on all the active **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP*** lines

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

where every active line is asserted. The sync word is followed by the data words. Data lengths of 2, 4, 8, 32, or 64 can be set up through the CSR mechanism and invoked with the data length, **DI[ ]***, command s nal. After the data are transmitted, a longitudinal even parity word shall be sent. Even parity is calculated a per serial bit basis over all the bits sent, including the sync and parity bits. A packet with correct parity v always finish transmission with all of its **AD[ ]***, **D[ ]***, **BP[ ]***, **TG[ ]***, and **TP*** lines in the released stat

6.2.3.22 Master disconnection phase

1) The MASTER shall wait until DISCONNECTION_PHASE | TRANSACTION_TIMEOUT.
2) If ~COMPELLED_STATUS the MASTER shall wait until the parity bit of the last queued packet transferred or until TRANSACTION_TIMEOUT is set.
3) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recover as specified in 6.2.3.24.
4) If WRITE_CMD | SPLIT_STATUS the MASTER shall activate **AD[ ]*** and **BP[ ]*** with the appropriate disconnection information.
5) If WRITE_CMD | SPLIT_STATUS is set the MASTER shall activate **tg[ ]*** and **tp*** with appropriate information.
6) The MASTER shall activate **cm[ ]*** and **cp*** with the appropriate information.
7) The MASTER shall release **as*** and assert **ai*.**
8) If COMPELLED_STATUS & **ds*** the MASTER shall wait until ~BROADCAST_STATUS & ~**DK*** | BROADCAST_STATUS & ~**DKf** | TRANSACTION_TIMEOUT.
9) If TRANSACTION_TIMEOUT is set the MASTER shall proceed to transaction timeout recover as specified in 6.2.3.24.
10) The MASTER shall release **ds*** and **ak***.
11) Wait until ~**AKf**.
12) Release **ad[ ]***, **d[ ]***, **bp[ ]***, **tg[ ]***, **tp***, **cm[ ]***, and **cp***.
13) If **TE*** is released update appropriate module state information.
14) Then set TRANSACTION_COMPLETE.
15) If the MASTER chooses to perform another transaction, proceed to the connection phase as specified in 6.2.3.1.
16) Release **et***.

6.2.3.23 Slave disconnection phase

1) If ~COMPELLED_STATUS wait until the parity bit of the last queued packet is transferred.
2) If WRITE_CMD | SPLIT_STATUS and the slave intends to use the data, a slave shall evaluate **AD[ ]*, BP[ ]*, TG[ ]*, TP[ ]***.
3) If the slave intends to use the data, evaluate **CM[ ]*** and **CP***.
4) Assert **ai***.
5) If asserted, release **dk***. If asserted, release **di***.
6) If ~LOCK_HOLD clear RESOURCE_LOCK.
7) If an error was detected during the transaction assert **te***.
8) Release **ak***.
9) Wait until DISCONNECTION_END.
10) Release **st[7..1]*** and **ca[ ]***.
11) If **TE*** is released update appropriate module state information.
12) Then set TRANSACTION_COMPLETE.

6.2.3.24 Transaction timeout recovery

When TRANSACTION_TIMEOUT sets, the MASTER shall execute a bus initialize operation as specified in 7.2.3.3. Isolation and recovery from the error that caused the transaction timeout is the responsibility of the system designer.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7. Bus/system management

7.1 Description

7.1.1 Bus control

Table 13 summarizes the relationships between the length of time **RE*** is asserted and the resulting events. Details are provided in the clauses that follow.

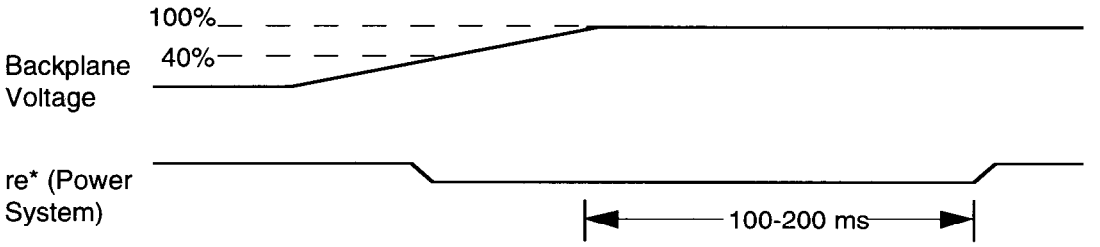
Table 13—Event and RE* relationships

Event	Event is detected if RE* is asserted for the following:	To cause an event, assert RE* for the following:
Power up	30 ms (and internal state shows power up)	100–200 ms (plus some self test)
System reset	30–500 ms	100–200 ms
Bus initialize	2–30 ms	4–12 ms
Live insertion alignment	1–129 μs	1 μs after current transaction completes
Stuck line	System or profile dependent*	Not applicable

*See fault tolerant profile: IEEE P896.9 (except during power up). IEEE P896.9 is an authorized standards project; for information on the project contact the IEEE Standards Department, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331 USA.

7.1.1.1 Power-up

When a system is first powered, the power system asserts the reset signal, **re***. The power system must assert **re*** prior to any of the bus power voltages reaching 40% of their nominal value. The power system maintains **re*** asserted for 100–200 ms after all of the bus power voltages are within regulation. This allows all modules on the bus to detect that a system reset is occurring. This power-up sequence is shown in figure 33.



NOTE—The bus voltage is shown as a single positive voltage for illustration purposes only. Actual bus voltages are specified in other standards and may be positive or negative or both.

Figure 33—Power-up sequence

7.1.1.2 System reset

Any module initiating a system reset asserts **re*** for 100–200 ms. All modules in a system monitor the **RE*** signal. If a module detects that **RE*** is asserted for more than 30 ms, a system reset is occurring. When a module detects that a system reset is occurring, it asserts **re***. The module then performs the internal reset

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

operations necessary to allow participation in bus transactions. When a module has completed these operations it releases **re***, indicating that it is ready to participate in transactions. After releasing **re***, modules wait until they detect **REf** at logic zero. This indicates that all modules are ready to allow bus operations.

Internal reset operations may include self test or other operations that may take a long time. It may be desirable to design modules that release **re*** following the time that they are fully operational but before they have completed self test. In this case, the modules need only be capable of asserting busy status in transactions subsequent to the release of **re*** but prior to the completion of self test. A system reset sequence is shown in figure 34.

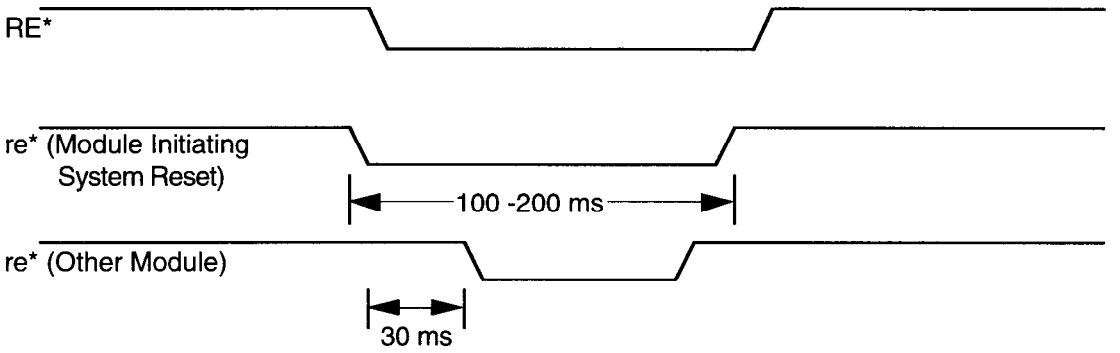


Figure 34—System reset sequence

7.1.1.3 Bus initialize

If modules detect **RE*** asserted for more than 2 ms a bus initialize operation is occurring. When a module detects a bus initialize operation is occurring it initializes its bus interface circuitry and waits to determine if it is also a system reset. If the module detects **REf** at logic zero prior to 30 ms it resumes operation. Bus initialize resets the bus interface but should be transparent to the rest of the system.

7.1.1.4 Live insertion

The logical protocols of Futurebus+ provide facilities for modules that are inserted into a powered operating system to align their logic state machines with the other modules on the bus and begin operation. When a module is inserted into the bus and detects that it has just powered up, it checks to see if **RE*** is asserted. If it is not asserted, the module assumes it has been inserted into a live system. If the reset line is asserted, it waits to see if it is a full system reset and not a bus initialize or live insertion. Eventually the module knows if it has been inserted into a live system.

A module inserted into a live system asserts **re*** to notify all the other modules that it needs to align. Modules do not initiate any new transactions, which causes the bus to go into an idle state. The inserted module waits after it has asserted reset until it has detected the bus in a continuous idle state for at least 1 μs and then it asserts **ai*** and **ar*** to complete the alignment. It then releases **re*** and all modules resume operation when they detect **REf** at logic zero.

After the alignment sequence, all the modules on the bus are cognizant of the live insertion.

7.1.1.5 Live withdrawal

The logical protocols of Futurebus+ also provide facilities for modules to be removed from powered operating systems without disturbing the operation of other modules. Modules to be withdrawn must release any signals they are asserting in such a way that other modules do not mistake the release for normal operation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The operator removing the module must first notify the board through some mechanism that it is about to be removed. The module, if it is able, then completes any tasks it has currently in progress. It then may be required to copy internally stored information to other locations in the system. When it has completed the transactions required to accomplish these tasks, the module must then inhibit itself from participating in any more bus transactions. At that time the module should only be asserting **ai*** and/or **ak*** and possibly one or more arbitrated message synchronization lines.

Modules are allowed to release **ak*** during the data phase or to release **ai*** if the parallel bus is idle. Modules are allowed to withdraw from the arbitrated message protocol in one of three points. During phase 0 a module is allowed to withdraw by releasing **ar***. During phase 2 a module is allowed to withdraw by releasing **ap***. During phase 4 a module is allowed to withdraw by releasing **aq***.

7.1.2 Futurebus+ control and status registers

A Futurebus+ module is required to provide three types of control and status registers. Module capability registers contain information indicating the capabilities a particular module has implemented. Module control registers are used to enable capabilities and change control parameters that affect bus operation. Module status registers are primarily used to indicate error and other status information. Only a limited number of these registers are specified, but space is reserved for others that are defined and specified in other standards. Registers are assumed to be 32-bit quantities. The physical addresses of the registers are specified in higher level standards.

7.1.2.1 Module capability registers

All modules implement the following capability registers:

— MODULE_LOGICAL_CAPABILITY
— COMPETITION_INTERNAL_DELAY
— MESSAGE_FRAME_SIZE
— PACKET_SPEED

Module capability registers are read only. In most systems, the data for these registers will reside in a capabilities ROM.

7.1.2.1.1 Module logical capability register

The module logical capability register is shown in figure 35. The individual fields are described in the following paragraphs:

The PACKET_LENGTH_n bits indicate which packet lengths the module supports. Legal packet lengths are 2, 4, 8, 16, 32, and 64 transfers.

The MULTIPLE_PACKET_CAPABLE bit, when set, indicates that the module supports the multiple packet transfer mode.

The PACKET_CAPABLE bit, when set, indicates if the module supports packet mode.

The TAG_CAPABLE bit indicates that a module has implemented one or more tag lines. The number of tag lines field indicates how many tag lines the module has implemented.

The FETCH_ADD_CAPABLE bit indicates if the module is capable of responding to FETCH_ADD locked commands.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

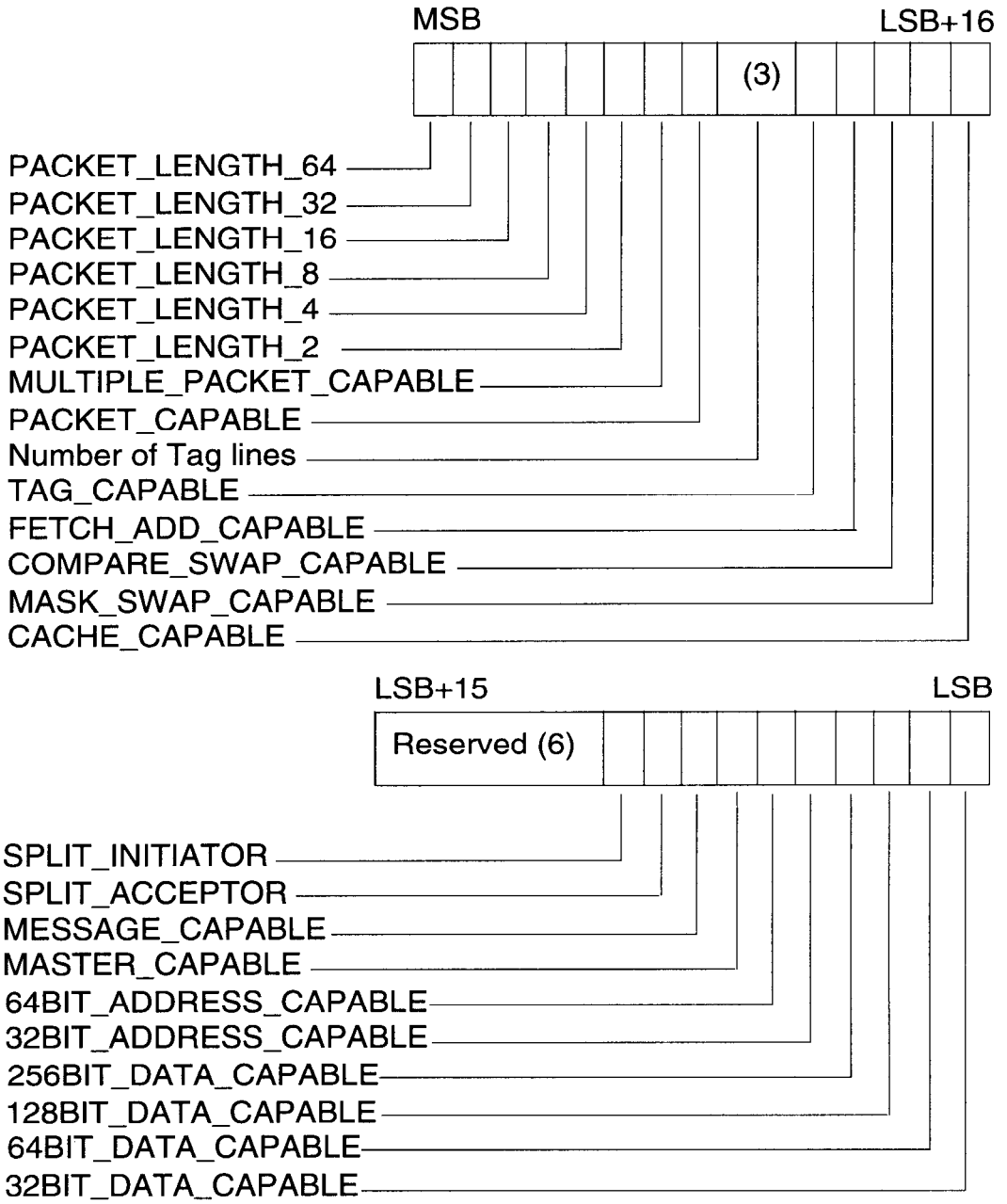


Figure 35—Module logical capability register

The COMPARE_SWAP_CAPABLE bit indicates if the module is capable of responding to COMPARE_SWAP locked commands.

The MASK_SWAP_CAPABLE bit indicates if the module is capable of responding to MASK_SWAP locked commands.

The CACHE_CAPABLE bit indicates if the module is capable of participating in transactions to cache coherent shared memory.

The SPLIT_INITIATOR bit indicates that a module is capable of initiating split transactions. A split initiator is capable of asserting **SR*** and later performing a response bus transaction.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The SPLIT_ACCEPTOR bit indicates that a module is capable of accepting split transactions. When a split acceptor is master and performing a transaction, it is capable of detecting **SR*** asserted, ending its transaction, and waiting for a response transaction from the module which asserted **SR***.

The MESSAGE_CAPABLE bit indicates the module is capable of participating in message-passing transactions as specified in 9.2.

The MASTER_CAPABLE bit indicates that a module is capable of bus mastership.

The 64BIT_ADDRESS_CAPABLE bit indicates that the module supports a 64 bit address width in addition to the default 32 bit address width.

The 32BIT_ADDRESS_CAPABLE bit must be set, as it indicates that the module supports a 32 bit address width, which all modules must support.

The 32BIT_DATA_CAPABLE, 64BIT_DATA_CAPABLE, 128BIT_DATA_CAPABLE, and 256BIT_DATA_CAPABLE bits indicate which data widths the module supports. The 32BIT_DATA_CAPABLE bit must be set since all modules must support this width.

7.1.2.1.2 Competition internal delay register

The system uses the competition internal delay register (see figure 36) to calculate the worst-case competition arbitrated message bus settling time.

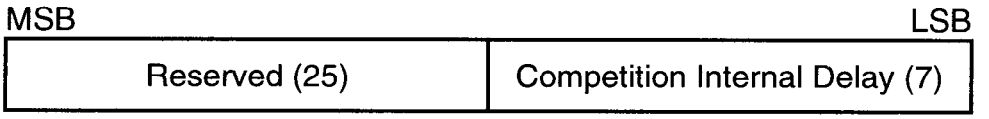


Figure 36—Competition internal delay register

7.1.2.1.3 Message frame size register

All message-capable modules must be able to pass message frames of 64 bytes. A message-capable module can indicate the ability to pass messages whose size is an integer multiple of 64 bytes. The maximum message size is indicated by the value in the message frame size register (see figure 37).

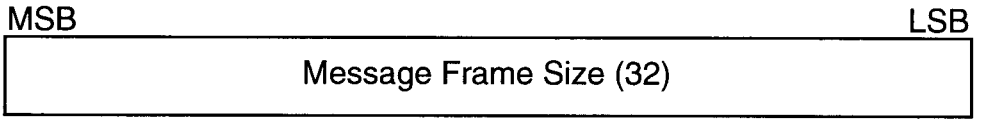


Figure 37—Message frame size register

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.1.2.1.4 Packet speed register

The packet speed register (see figure 38) contains four 8-bit fields indicating one to four packet transfer speeds supported by the module. A value of zero indicates the field doesn't contain a supported packet transfer speed.

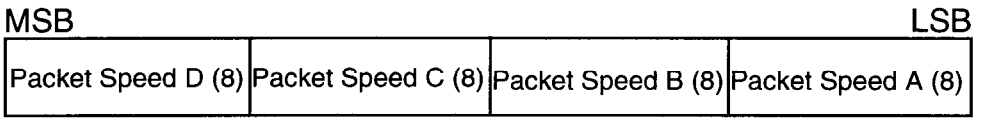


Figure 38—Packet speed register

7.1.2.2 Module control registers

All modules implement the following control registers:

— LOGICAL_COMMON_CONTROL_REGISTER
— LOGICAL_MODULE_CONTROL_REGISTER
— BUS_PROPAGATION_DELAY_REGISTER
— COMPETITION_SETTLING_TIME_REGISTER
— TRANSACTION_TIMEOUT_REGISTER
— MESSAGE_PASSING_SELECT_MASK

Module control registers are read/write.

7.1.2.2.1 Logical common control register

The logical common control register is shown in figure 39. The individual fields are described in the following paragraphs:

The configuration process sets the CENTRAL_ARBITER bit to select the central arbiter instead of the distributed arbiter for the purpose of allocating bus mastership. This bit is also be set by hardware when **REf** transitions to logic zero following a power up or system reset if a central arbiter is present in the backplane.

The configuration process sets the MULTIPLE_PACKET_MODE_ENABLE bit to indicate that the module may generate transactions using multiple packet mode.

The configuration process sets the PACKET_LENGTH_n bits to indicate which packet transfer lengths the module is allowed to use.

The configuration process sets the high packet speed field to one of four values to indicate the packet speed the module may use during high speed packet transfers.

The configuration process sets the low packet speed field to one of four values to indicate the packet speed the module may use during low speed packet transfers.

The configuration process sets the DISTRIBUTED_MESSAGE_ENABLE bit to allow the module to send distributed arbitration messages.

The configuration process sets the SPLIT_ENABLE bit to indicate that the module may split bus transactions if it has the capability.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

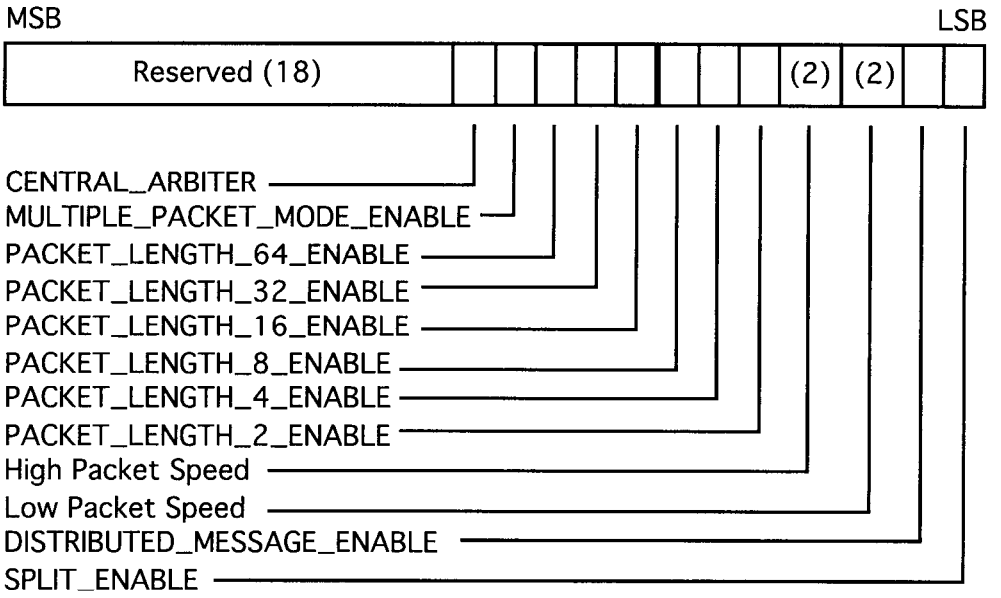


Figure 39—Logical common control register

7.1.2.2.2 Logical module control register

The logical module control register is shown in figure 40. The individual fields are described in the following paragraphs:

The configuration process sets the TAG_ENABLE bit to indicate that tag bits are active and that the module may activate tag bits and must check tag parity.

The configuration process sets the PARITY_REPORT_ENABLE bit to indicate that modules are to report parity errors by asserting **be*** and **te***.

The configuration process sets the HIGH_SPEED_ENABLE bit to indicate that the module is capable of high speed packet operation.

The configuration process sets MASTER_ENABLE to enable a module's master capability.

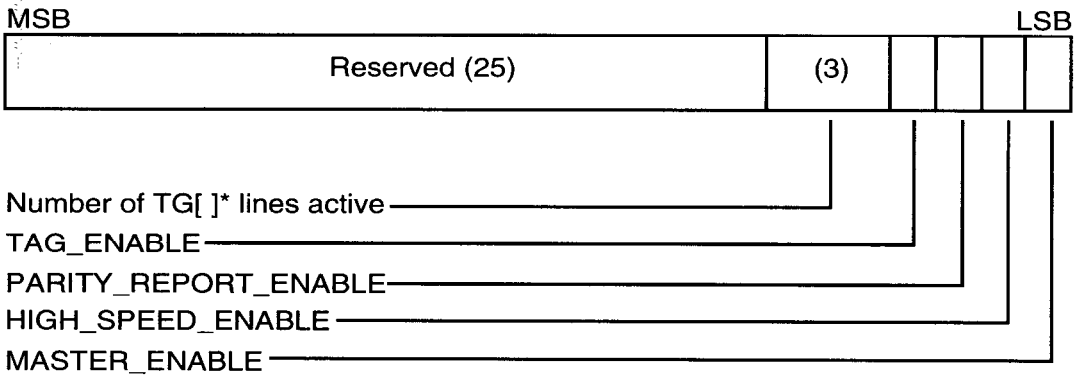


Figure 40—Logical module control register

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.1.2.2.3 Bus propagation delay register

(See figure 41.) The configuration process sets the bus propagation delay field to the value of the one-way backplane propagation delay. A module may use it to set its wire-OR integrator and arbitrated message delay.

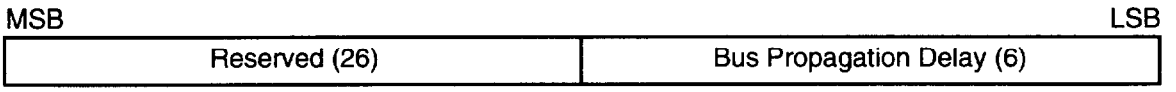


Figure 41—Bus propagation delay register

7.1.2.2.4 Competition settling time register

(See figure 42.) The configuration process sets the competition settling time register to the worst-case value that a module is to use to determine its competition settling time during arbitrated message competition.

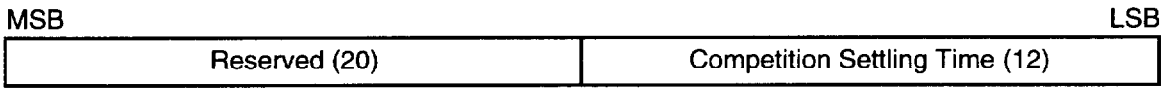


Figure 42—Competition settling time register

7.1.2.2.5 Transaction timeout register

(See figure 43.) The configuration process sets the transaction timeout field to the amount of time a master is to wait before its transaction timeout attribute sets.



Figure 43—Transaction timeout register

7.1.2.2.6 Message-passing select mask

(See figure 44.) Bits in the message-passing select mask are set to enable modules to receive certain messages through their broadcast mailbox. The least significant bit is unused.



Figure 44—Message-passing select mask

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2 Specification

7.2.1 Bus control attributes

BUS_HOLD. Modules shall set BUS_HOLD if **RE*** & ~BUS_INIT & ~SYS_RESET & ~POWER_UP & ~LIVE_INSERTION. Modules shall maintain BUS_HOLD set until LIVE_INSERTION | BUS_INIT | SYS_RESET.

BUS_IDLE. Modules shall set BUS_IDLE if **~AQf** & **AR*** & **AI*** &**~AKf**.

BUS_IDLE_1US. Modules shall set BUS_IDLE_1US if **re*** & (BUS_IDLE has been set for a minimum of 1 μs) and shall maintain it until ~BUS_IDLE.

BUS_INIT. Modules shall set BUS_INIT if **REf** & ~SYS_RESET & ~POWER_UP for a minimum of 2 ms. Modules shall maintain BUS_INIT set until SYS_RESET | POWER_UP | **~REf**.

ENTRANT. Modules shall set ENTRANT if POWER_UP & **~REf**. Modules shall maintain ENTRANT set until **ai*** & **ar***.

INIT. Modules shall set INIT if POWER_UP | SYS_RESET | BUS_INIT and shall maintain INIT until the module is sufficiently initialized to participate in basic bus transactions (respond BUSY as a minimum).

LIVE_INSERTION. Modules shall set LIVE_INSERTION if ~POWER_UP & ~ENTRANT & RESET_EVENT & **~REf**. Modules shall maintain LIVE_INSERTION set until **AS*** is asserted.

LIVE_WITHDRAWAL_COMMAND. LIVE_WITHDRAWAL_COMMAND shall be set on a module when the module is notified it is to be withdrawn.

POWER_UP. Modules shall set POWER_UP when power is applied. Live inserted modules shall not assert any bus signals prior to POWER_UP setting. Modules shall maintain POWER_UP set until SYS_RESET | BUS_IDLE_1US & **ai*** & **ar***. Modules shall wait a minimum of 1 μs after asserting **re*** prior to clearing POWER_UP. Modules not capable of driving **ar*** may assume it is asserted.

POWER_UP_TIME. Modules shall set POWER_UP_TIME when power is applied. Modules shall clear POWER_UP_TIME a minimum 500 ms and a maximum 1000 ms later.

READY_FOR_WITHDRAWAL. Modules shall set READY_FOR_WITHDRAWAL if LIVE_WITHDRAWAL_COMMAND & **~ap*** & **~aq*** & **~ar*** & **~as*** & **~ak*** & **~ai*** & **~cm[ ]*** & **~cp*** & **~st[ ]*** & **~ca[ ]*** & **~re*** & **~ab[ ]*** & **~abp*** & **~ac0*** & **~ac1*** & **~ad[ ]*** & **~d[ ]*** & . **~bp[ ]*** & **~rq0*** & **~rq1*** & **~et***.

RESET_COMPLETE. A module shall set RESET_COMPLETE if SYS_RESET and it has completed its reset procedure and is ready to respond to bus transactions. Modules shall maintain RESET_COMPLETE set until **~REf**.

RESET_EVENT. Modules shall set RESET_EVENT if **RE***. Modules shall maintain RESET_EVENT set until SYS_RESET | BUS_INIT | LIVE_INSERTION.

SYS_RESET. Modules shall set SYS_RESET when **REf** has been asserted for a minimum of 30 ms. Modules shall maintain SYS_RESET set until **~REf**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.2 RE* reset signal

Modules shall assert **re*** if SYS_RESET & ~RESET_COMPLETE | ENTRANT & **~REf** or to initiate a system reset, bus initialize, or live insertion as specified in 7.2.3. Modules shall maintain **re*** asserted until SYS_RESET & RESET_COMPLETE | ENTRANT & BUS_IDLE_1US & **ar*** & **ai***. Modules shall release **RE*** no later than 1 μs after ~(SYSTEM_RESET & RESET_COMPLETE | ENTRANT & BUS_IDLE_1US & **ar*** & **ai***) is true.

7.2.3 Protocol definition

7.2.3.1 Power-up

The power system shall

1) Prior to any bus voltage reaching 40% of nominal voltage assert **re***.
2) Maintain **re*** asserted until all bus voltages are within regulation specifications.
3) Then continue to assert **re*** for an additional 100–200 ms.
4) Then release **re***.

Modules shall

1) Upon detecting the presence of power and becoming capable, set POWER_UP.
2) Wait until SYS_RESET | **~REf**.
3) If SYS_RESET proceed to system reset as specified in 7.2.3.2.
4) If **~REf** proceed to live insertion as specified in 7.2.3.4.

7.2.3.2 System reset

Modules upon detecting SYS_RESET shall

1) Assert **re***.
2) Perform all internal reset operations necessary to allow participation in bus transactions.
3) Release **re***.
4) Wait until **~REf** or 500–1000 ms & POWER_UP_TIME.
5) Proceed with bus operations as requested.
6) To initiate a system reset operation a module shall assert **re*** for 100–200 ms and then proceed to 2).

7.2.3.3 Bus initialize

Modules upon detecting BUS_INIT shall

1) Initialize all bus interface circuitry.
2) Wait until ~BUS_INIT.
3) Proceed with bus operations, if ~SYS_RESET, as requested.

To initiate a bus initialize operation a module shall assert **re*** for a minimum of 4 ms and not longer than 12 ms and then proceed to 2).

7.2.3.4 Live insertion

Modules shall

1) Perform all internal reset operations necessary to allow participation in bus transactions.
2) If **RE*** wait until SYS_RESET | **~REf**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3) If SYS_RESET proceed to system reset as specified in 7.2.3.2.
4) If ~**REf** wait for a minimum of 1 μs and then assert **re***.
5) Wait for BUS_IDLE_1US | SYS_RESET.
6) If SYS_RESET proceed to system reset as specified in 7.2.3.2.
7) If BUS_IDLE_1US assert **ai*** and if capable assert **ar***.
8) Release **re***.
9) Proceed with bus operations as requested.

7.2.3.5 Live withdrawal

Modules upon detecting LIVE_WITHDRAWAL_COMMAND shall

1) Perform any system-dependent save state operations.
2) Wait for the necessary events in a)–e) below and withdraw the appropriate signals until ~**ap*** & ~**aq*** & ~**ar*** & ~**as*** & ~**ak*** & ~**ai*** & ~**cm[ ]*** & ~**cp*** & ~**st[ ]*** & ~**ca[ ]*** & ~**re*** & ~**ab[ ]*** & ~**abp*** & ~**ac0*** & ~**ac1*** & ~**ad[ ]*** & ~**d[ ]*** & ~**bp[ ]*** & ~**rq0*** & ~**rq1*** & ~**et*** is true.
   a) After detecting the assertion of **AS*** modules shall not assert **ak*** and if **AS*** & **AK*** & ~**ak*** shall release **ai***.
   b) After detecting the release of **AS*** modules shall not assert **ai*** and if ~**AS*** & **AI*** & ~**ai*** release **ak***.
   c) If ~**AQf** & **ar*** & ~**ap*** release **ar***.
   d) If ~**ARf** & **ap*** &~**aq*** release **ap***.
   e) If ~**APf** & **aq*** & ~**ar*** release **aq***.
3) Set READY_FOR_WITHDRAWAL.

7.2.4 Futurebus+ control and status registers

Fields that are marked "reserved" *shall* return all zeros on a "read." Data "writes" to "reserved" fields *shall* have the value of zero. All implemented bits of control registers *shall* return what is written.

All timer registers are programmed in units of 2^{-32} s (~233 ps). Timer resolution is not implied by this register format.

7.2.4.1 Module capability registers

Modules shall implement the following capability registers:

— MODULE_LOGICAL_CAPABILITY
— COMPETITION_INTERNAL_DELAY
— MESSAGE_FRAME_SIZE
— PACKET_SPEED

Module capability registers shall be read only.

7.2.4.1.1 Module logical capability register

PACKET_LENGTH_64_CAPABLE. The LSB+31 shall be PACKET_LENGTH_64_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol with TRANSFER_64.

PACKET_LENGTH_32_CAPABLE. The LSB+30 shall be PACKET_LENGTH_32_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol with TRANSFER_32.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

PACKET_LENGTH_16_CAPABLE. The LSB+29 shall be PACKET_LENGTH_16_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol with TRANSFER_16.

PACKET_LENGTH_8_CAPABLE. The LSB+28 shall be PACKET_LENGTH_8_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol with TRANSFER_8.

PACKET_LENGTH_4_CAPABLE. The LSB+27 shall be PACKET_LENGTH_4_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol with TRANSFER_4.

PACKET_LENGTH_2_CAPABLE. The LSB+26 shall be PACKET_LENGTH_2_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol with TRANSFER_2.

MULTIPLE_PACKET_CAPABLE. The LSB+25 shall be MULTIPLE_PACKET_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol in multiple packet mode.

PACKET_CAPABLE. The LSB+24 shall be PACKET_CAPABLE and shall be set if the module is capable of participating in transactions using the packet protocol.

NUMBER OF TG[ ]* LINES. LSB+21, LSB+22, and LSB+23 shall be a three bit field containing the number of **TG[ ]*** lines a module has implemented. A value of zero shall correspond to one **TG[ ]*** line and a value of seven shall correspond to eight **TG[ ]*** lines.

TAG_CAPABLE. The LSB+20 shall be TAG_CAPABLE and shall be set if the module has one or more **TG[ ]*** lines implemented.

FETCH_ADD_CAPABLE. The LSB+19 shall be FETCH_ADD_CAPABLE and shall be set if the module is capable of responding to both the FETCH_ADD_BIG and FETCH_ADD_LITTLE locked commands.

COMPARE_SWAP_CAPABLE. The LSB+18 shall be COMPARE_SWAP_CAPABLE and shall be set if the module is capable of responding to COMPARE_SWAP locked commands.

MASK_SWAP_CAPABLE. The LSB+17 shall be MASK_SWAP_CAPABLE and shall be set if the module is capable of responding to MASK_SWAP locked commands.

CACHE_CAPABLE. The LSB+16 shall be CACHE_CAPABLE and shall be set if the module is capable of performing transactions in accordance with 8.2.

SPLIT_INITIATOR. The LSB+9 shall be SPLIT_INITIATOR and shall be set if the module is capable of initiating split transactions.

SPLIT_ACCEPTOR. The LSB+8 shall be SPLIT_ACCEPTOR and shall be set if the module is capable of accepting split transactions.

MESSAGE_CAPABLE. The LSB+7 shall be MESSAGE_CAPABLE and shall be set if the module is capable of participating in transactions using the message-passing protocol.

MASTER_CAPABLE. The LSB+6 shall be MASTER_CAPABLE and shall be set if the module is capable of becoming MASTER.

64BIT_ADDRESS_CAPABLE. The LSB+5 shall be 64BIT_ADDRESS_CAPABLE and shall be set if the module is capable of participating in transactions with a 64-bit address path width.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

32BIT_ADDRESS_CAPABLE. The LSB+4 shall be 32BIT_ADDRESS_CAPABLE and shall be set.

256BIT_DATA_CAPABLE. The LSB+3 shall be 256BIT_DATA_CAPABLE and shall be set if the module is capable of participating in transactions with a 256-bit data path width.

128BIT_DATA_CAPABLE. The LSB+2 shall be 128BIT_DATA_CAPABLE and shall be set if the module is capable of participating in transactions with a 128-bit data path width.

64BIT_DATA_CAPABLE. The LSB+1 shall be 64BIT_DATA_CAPABLE and shall be set if the module is capable of participating in transactions with a 64-bit data path width.

32BIT_DATA_CAPABLE. The LSB shall be 32BIT_DATA_CAPABLE and shall be set.

7.2.4.1.2 Competition internal delay register

The least significant 7 bits of the competition internal delay register shall correspond to the worst-case delay between adjacent pairs of the arbitrated message bus lines through the modules competition logic as follows:

1) **AB7*** to **AB6***
2) **AB6*** to **AB5***
3) **AB5*** to **AB4***
4) **AB4*** to **AB3***
5) **AB3*** to **AB2***
6) **AB2*** to **AB1***
7) **AB1*** to **AB0***
8) **AB0*** to **ABP***

The field shall be in steps of 2^{-32} s (~233 ps). Note that the maximum value of this field is less than 30 ns.

7.2.4.1.3 Message frame size register

All message capable modules shall be able to transmit and receive message frames of 64 bytes. The value in the message frame size register shall indicate the maximum number of bytes that a message-passing capable module can pass in a single message frame. The value in the message frame size register shall equal 0 Mod 64, that is the least significant six bits shall be zero.

7.2.4.1.4 Packet speed register

The packet speed register contains four 8-bit fields specifying one to four packet transfer speeds supported by the module. The transfer speed shall be defined as the reciprocal of one bit period. The value in each field is in steps of 1 MHz. A value of zero indicates the field does not contain a supported packet transfer speed.

The field comprised of the LSB through the LSB+7 inclusive corresponds to Packet Transfer Speed A.

The field comprised of LSB+8 through the LSB+15 inclusive corresponds to Packet Transfer Speed B.

The field comprised of LSB+16 through the LSB+23 inclusive corresponds to Packet Transfer Speed C.

The field comprised of LSB+24 through the MSB inclusive corresponds to Packet Transfer Speed D.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.4.2 Module control registers

Modules shall implement the following control registers as specified in a higher level standard, such as IEEE Std 896.2-1991:

— Logical common control register
— Logical module control register
— Bus propagation delay register
— Competition settling time register
— Transaction timeout register
— Message-passing select mask register

7.2.4.2.1 Logical common control register

CENTRAL_ARBITER. The LSB+13 shall be CENTRAL_ARBITER. Modules shall use the central arbiter for the purpose of parallel bus allocation if this bit is set. Modules shall use the distributed arbiter for this purpose if this bit is not set. Modules shall set CENTRAL_ARBITER if **PE*** is asserted or clear CENTRAL_ARBITER if **PE*** is released when they detect **REf** transitioning to logic zero following POWER_UP | SYS_RESET.

MULTIPLE_PACKET_MODE_ENABLE. The LSB+12 shall be MULTIPLE_PACKET_MODE_ENABLE. Modules capable of multiple packet mode may initiate multiple packet transfers if MULTIPLE_PACKET_MODE_ENABLE is set, and one of the following PACKET_LENGTH_n_ENABLE bits is set. Modules shall clear MULTIPLE_PACKET_MODE_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

PACKET_LENGTH_64_ENABLE. The LSB+11 shall be PACKET_LENGTH_64_ENABLE. Modules may initiate packet transfers if PACKET_LENGTH_64_ENABLE is set. Modules shall clear PACKET_LENGTH_64_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

PACKET_LENGTH_32_ENABLE. The LSB+10 shall be PACKET_LENGTH_32_ENABLE. Modules may initiate packet transfers if PACKET_LENGTH_32_ENABLE is set. Modules shall clear PACKET_LENGTH_32_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

PACKET_LENGTH_16_ENABLE. The LSB+9 shall be PACKET_LENGTH_16_ENABLE. Modules may initiate packet transfers if PACKET_LENGTH_16_ENABLE is set. Modules shall clear PACKET_LENGTH_16_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

PACKET_LENGTH_8_ENABLE. The LSB+8 shall be PACKET_LENGTH_8_ENABLE. Modules may initiate packet transfers if PACKET_LENGTH_8_ENABLE is set. Modules shall clear PACKET_LENGTH_8_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

PACKET_LENGTH_4_ENABLE. The LSB+7 shall be PACKET_LENGTH_4_ENABLE. Modules may initiate packet transfers if PACKET_LENGTH_4_ENABLE is set. Modules shall clear PACKET_LENGTH_4_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

PACKET_LENGTH_2_ENABLE. The LSB+6 shall be PACKET_LENGTH_2_ENABLE. Modules may initiate packet transfers if PACKET_LENGTH_2_ENABLE is set. Modules shall clear PACKET_LENGTH_2_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

HIGH PACKET SPEED. The LSB+4 and LSB+5 shall be a two bit field specifying the packet speed the module shall use if ~LOW_SPEED_STATUS. A value of zero corresponds to packet speed A. A value of three corresponds to packet speed D. Modules shall clear this field if SYS_RESET | BUS_INIT | POWER_UP.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

LOW PACKET SPEED. The LSB+2 and LSB+3 shall be a two bit field specifying the packet speed the module shall use if LOW_SPEED_STATUS. A value of zero corresponds to packet speed A. A value of three corresponds to packet speed D. Modules shall clear this field if SYS_RESET | BUS_INIT | POWER_UP.

DISTRIBUTED_MESSAGE_ENABLE. The LSB+1 shall be DISTRIBUTED_MESSAGE_ENABLE. Modules shall clear DISTRIBUTED_MESSAGE_ENABLE if SYS_RESET | BUS_INIT | POWER_UP. If this bit is cleared by the monarch, modules shall withdraw from an active arbitration competition by clearing DISTRIBUTED_MESSAGE_REQUEST and CENTRAL_MESSAGE_REQUEST and ARBITRATED_MESSAGE_REQUEST as specified in 5.2.1 and 5.2.2.

SPLIT_ENABLE. The LSB shall be SPLIT_ENABLE. Modules may split bus transactions if SPLIT_ENABLE is set. Modules shall clear SPLIT_ENABLE if SYS_RESET | BUS_INIT | POWER_UP. The configuration process shall not set SPLIT_ENABLE unless all modules have the SPLIT_ACCEPTOR capability bit set.

7.2.4.2.2 Logical module control register

NUMBER OF TAG LINES. The LSB+4, LSB+5, and LSB+6 shall be a three-bit field specifying the number of **TG[ ]*** lines that shall be active. A value of 0 shall correspond to one tag line, **TG0***, and a value of 7 will correspond to eight tag lines, **TG[7..0]***.

TAG_ENABLE. The LSB+3 shall be TAG_ENABLE. Modules shall enable parity checking on **TG[ ]*** if TAG_ENABLE is set. Modules shall clear TAG_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

PARITY_REPORT_ENABLE. The LSB+2 shall be PARITY_REPORT_ENABLE. Modules shall assert **be*** and **te*** in response to parity errors if PARITY_REPORT_ENABLE is set. Modules shall clear PARITY_REPORT_ENABLE if SYS_RESET | BUS_INIT | POWER_UP.

HIGH_SPEED_ENABLE. The LSB+1 shall be HIGH_SPEED_ENABLE. Modules shall clear HIGH_SPEED_ENABLE if SYS_RESET | BUS_INIT | POWER_UP. The configuration process may set HIGH_SPEED_ENABLE on a given module if it determines a high and a low packet speed for the bus segment and if on the module the high packet speed is supported. The configuration process shall initialize the high packet speed field of the Logical Common Control Register to point to the bus-wide high packet speed before setting HIGH_SPEED_ENABLE. If HIGH_SPEED_ENABLE is clear, a module shall assert **ts*** (**ca0***) for packet mode transfers.

MASTER_ENABLE. The LSB shall be MASTER_ENABLE. If MASTER_ENABLE is set, a module may set either DISTRIBUTED_ARBITER_REQUEST or (rq0* | rq1*). Modules which are not monarch capable shall clear MASTER_ENABLE if SYS_RESET | BUS_INIT | POWER_UP. Modules which are monarch capable set and clear this bit as specified in the appropriate profile in IEEE Std 896.2-1991.

7.2.4.2.3 Bus propagation delay register

The least significant 6 bits of the bus propagation delay register shall be the maximum end-to-end signal propagation delay along the logical bus. The field shall be in steps of 2^{-32} s (~233 ps). The register shall be set to a value of $3F_{16}$ (14.7 ns) if SYS_RESET | BUS_INIT | POWER_UP.

Modules shall ensure that a change in this value does not cause any arbitrated message or parallel protocol violations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.4.2.4 *Competition settling time register*

The least significant 12 bits of the competition settling time register shall correspond to the worst-case value that the module uses to determine its competition settling time during arbitrated message transmission. $\mathbf{t_{pd''}}$ is the value in the bus propagation delay register as specified in 7.2.4.2.3. $\mathbf{t_{int}}$ is the value in the modules competition internal delay register as specified in 7.2.4.1.2. $\mathbf{t_{ext}}$ is the largest of the values in the competition internal delay registers of the other modules in the system. The following equation may be used to calculate the settling time:

$$\mathbf{t_a} = 10 * \mathbf{t_{pd''}} + 3 * \mathbf{t_{int}} + 5 * \mathbf{t_{ext}}$$

NOTE—Depending on the competition number used by the module, its slot position, and the implementation of the competition logic, the actual worst-case settling time can be significantly lower than the value given by this equation. The system shall program this register with a value equal to or greater than the true worst-case settling time required by the module.

The field shall be in steps of 2^{-32} s (~233 ps). The register shall be set to 600_{16} (1536_{10}), which is 358 ns, if SYS_RESET | BUS_INIT | POWER_UP.

7.2.4.2.5 Transaction timeout register

The transaction timeout register shall be the transaction timeout time used by the master to determine when to set TRANSACTION_TIMEOUT as specified in 6.2. The field shall be in steps of 2^{-32} s. The transaction timeout shall be set to 80000_{16} (524288_{10}), which is 122 μs, if SYS_RESET | BUS_INIT | POWER_UP. A value of zero shall disable the transaction timeout.

7.2.4.2.6 Message-passing select mask

The LSB+1 through LSB+63 bits in the message-passing select mask register are set to enable modules to receive certain messages through their broadcast mailbox as specified in 9.2. A zero shall always be written to the LSB. The LSB shall always read a zero.

The register shall be cleared if SYS_RESET | POWER_UP. Do NOT clear if BUS_INIT.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8. Cache coherence

8.1 Description

Futurebus+ includes facilities to allow multiple processor systems to be implemented using a shared memory architecture. An example of a shared memory architecture is shown in figure 45.

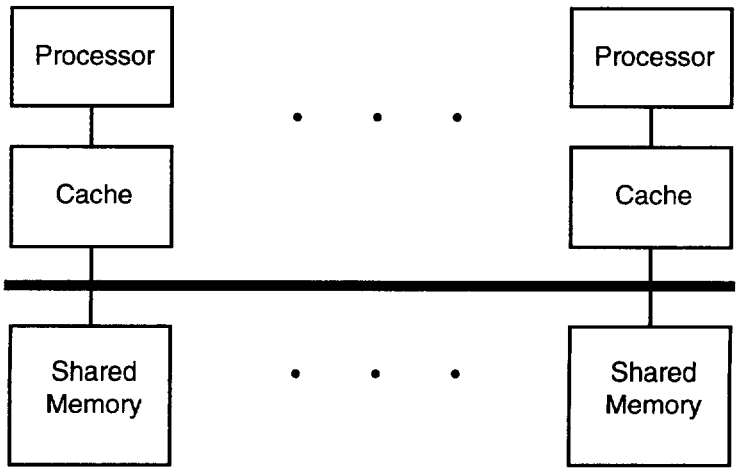


Figure 45—Shared memory architecture

The shared memory architecture consists of a number of processors that share a memory accessed using Futurebus+. All of the processors have an associated *cache memory* to accomplish three objectives:

1) To provide the processor with faster access to data
2) To reduce contention for use of the bus
3) To increase the efficiency of the bus

Almost any type of cache memory may be used to achieve the first objective. To achieve the second objective a *copyback cache* may be used. A copyback cache has the attribute that data written from a processor is only written to the cache. Data is written to the shared memory only if a cache line containing a modified entry is required in response to a cache miss. A copyback cache thereby reduces both the number of reads and the number of writes on the bus. To achieve the third objective a block of data is retrieved from memory as a result of a cache miss instead of just the word the processor requested. This turns a processors random accesses into efficient block transfers. This block of data is referred to as a *cache line*. The cache line size within a Futurebus+ system is fixed at 64 bytes. A cache line must be aligned modulus 64 in the byte address space and is always transferred in a single block transfer. Partial cache line transfers are not allowed.

A cache line is said to be *valid* if a cache has an up-to-date copy of the system image. A cache with a valid line has system permission to allow its processor to read the cache line privately. A cache with a valid line must inform the other modules of the existence of its copy. A cache line is either valid or invalid.

A cache line may be valid in multiple caches in the system. A cache line is said to be *exclusive* if the cache line is valid and if the cache has been guaranteed that there are no other valid cached copies in any other cache in the system. A cache with an exclusive line has system permission to assume the attribute of modified at any time. A valid cache line is either exclusive or *shared*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A cache line is said to be *modified* if the copy of the line in the cache supersedes the copy in shared memory. A cache with a modified line has system permission to allow its processor to read or write the cache line privately. A cache with a modified line must respond to transactions as an intervener. A cache with a modified line must update shared memory before it may relinquish the modified attribute. An exclusive cache line is either modified or unmodified.

The *cache coherence* protocol described and specified in this clause guarantees that all modules observe the modifications of a given cache line in the same order.

Not all addresses in a system are cacheable. The model shown in figure 47 is not intended to show packaging into boards, or other implementation details. Some implementations may have private memory in addition to shared memory, multiple processors per cache, multiple caches, all three module types on one board, and so on. The processor module can be any active user of memory, such as a microprocessor, multiple independent memory ports serving one processor, multiple independent microprocessors, a DMA based I/O controller, another hierarchical cache layer, and so on.

8.1.1 Cache attributes

The coherence of data in multiple caches is based on the cache coherence attributes *invalid, shared unmodified, exclusive unmodified*, and *exclusive modified*. Only one of these cache attributes may be valid for a cache line in a particular module at a given time. These four attributes comprise the cache line state.

The invalid attribute is true for a cache line if there is not an up-to-date copy in the module's cache. All cache lines in a module's cache are marked invalid when a system reset is received.

The shared unmodified attribute is true for a cache line if there is an up-to-date copy of the line in the module's cache and the module is to assume that a copy also exists in another module's cache. Modules are allowed to invalidate a shared unmodified copy of a cache line at any time.

The exclusive unmodified attribute is true for a cache line if there is an up-to-date copy of the line in the module's cache and the module has been guaranteed that no other copies of the line are valid in any other cache in the system.

The exclusive modified attribute is true for a cache line if there is an up-to-date copy of the line in the module's cache and the module has the only valid copy in the system. The module has the responsibility to eventually update memory.

8.1.2 Bus snooping

All caches are required to *snoop*, i.e., monitor, all cache coherent transactions on the system bus. In certain cases modules are required to perform cooperative actions during transactions to maintain cache coherence.

A cache with a cache line which is not invalid is required when it snoops certain transactions to change its cache line state.

A cache with an exclusive modified cache line is required to *intervene* in any read access of the modified location by asserting **iv*** and providing the cache line to the master in place of the memory. The cache must then change the state of the cache line to either shared unmodified or invalid.

A cache which snoops a read modified, invalidate, or write invalid transaction must change the state of the cache line to invalid.

In certain cases a cache is allowed to convert a transaction occurring on the bus to a broadcast operation and *snarf*, i.e., capture, the data. This allows the cache to obtain a shared copy of a cache line without an addi-

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

tional transaction. Modules may snarf read shared, copyback, read invalid, and shared response transactions. To snarf transactions modules assert **bc*** to convert the transaction to broadcast mode, **tf*** to notify other modules they are keeping a copy, and change the state of the cache line to shared unmodified.

If a cache has an exclusive modified copy of a cache line, and it wants to copy the line back to memory and it snoops a read transaction for the line, it must intervene in the transaction and supply the cache line. This allows the cache to invalidate the cache line and avoid the copyback transaction.

8.1.3 Cache coherence using connected transactions

8.1.3.1 State changes

This clause describes the state changes that are possible from each of the four coherence attributes.

As shown in figure 46, it is possible for a cache line to transition to any of the three other states from the invalid state.

A module that initiates a read shared transaction that completes successfully with **TF*** asserted will cause its cache line to change state to shared unmodified. If **TF*** is not asserted, the state will change to exclusive unmodified.

A module that asserts **bc*** and **tf*** to snarf a read shared, read invalid, or copyback transaction will cause its cache line to change state to shared unmodified.

A module that initiates a read modified transaction that completes successfully will cause its cache line to change state to exclusive modified.

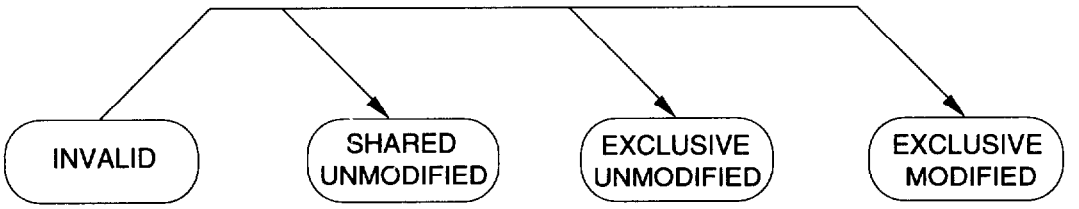


Figure 46—Invalid state

As shown in figure 47, it is possible for a cache line to transition to the invalid and exclusive modified states from the shared unmodified state.

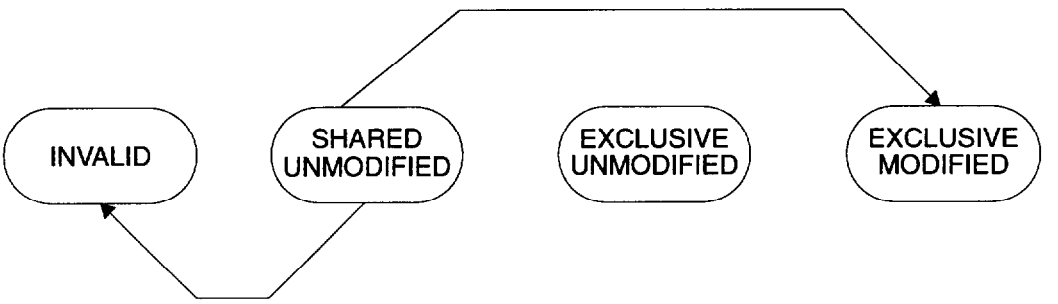


Figure 47—Shared unmodified state

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Any module may at any time change the state of a cache line from shared unmodified to invalid. It does this without using a bus transaction. If a module snoops a read shared or read invalid transaction and does not assert **tf***, it changes the state of its cache line to invalid. If a module snoops a read modified, write invalid, or invalidate transaction, it changes its cache line state to invalid.

A module that initiates an invalidate transaction that completes successfully will cause its cache line to change state to exclusive modified.

As shown in figure 48, it is possible for a cache line to transition to any of the three other states from the exclusive unmodified state.

Without using a bus transaction, any module may change the state of a cache line from exclusive unmodified to any of the other three states at any time.

A module that snoops a read shared or read invalid transaction in which it asserts **tf*** will cause its cache line to change state to shared unmodified. A module that snoops a read shared or read invalid transaction and does not assert **tf*** changes the state of its cache line to invalid. A module that snoops a read modified, write invalid, or invalidate transaction changes its cache line state to invalid.

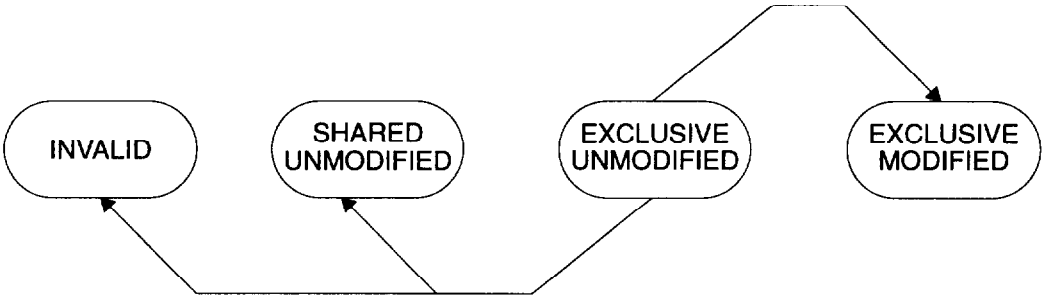


Figure 48—Exclusive unmodified state

As shown in figure 49, it is possible for a cache line to transition to the invalid or shared unmodified states from the exclusive modified state. A module can initiate a change from exclusive modified to invalid or shared unmodified by using a copy back transaction. If the module wishes to keep a copy of the cache line, it may transition to the shared unmodified state if it asserts **tf*** to notify other modules it is keeping the copy.

If a module snoops a read shared, read invalid, or read modified transaction, it must assert **iv*** to intervene and supply the data in the place of the memory. A module may keep a copy of the data if it asserts **tf*** during read shared and read invalid transactions.

Modules always change the state of a cache line to invalid if they snoop a write invalid transaction.

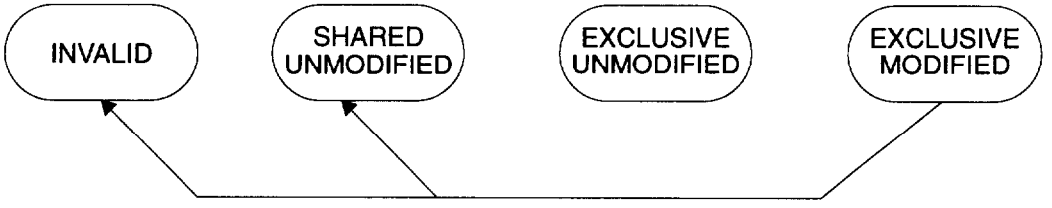


Figure 49—Exclusive modified state

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.1.3.2 Read miss process

When a processor issues a read, if the location requested is shared unmodified, exclusive unmodified, or exclusive modified, the operation is said to be a *read hit*. In the case of a read hit, no bus transaction is necessary and the processor is allowed immediate access to the data. If the location is invalid the operation is said to be a *read miss*.

When a read miss occurs, the cache arbitrates for the bus. When the cache obtains the bus, it issues a *read shared* transaction to request the cache line containing the requested data. When the transaction is complete, the data is given to the processor, stored in the cache, and the shared unmodified attribute is set for the cache line.

There is a special optimization for accesses to memory that is not actually shared. If no other cache signals that it contains the cache line by asserting **tf*** on a read shared transaction, the exclusive unmodified attribute may be set for the cache line. This allows the processor to update the cache line to exclusive modified without further bus transactions.

The exclusive unmodified attribute can only be set for a cache line, using connected transactions, by issuing a read shared transaction, and having it complete without **TF*** asserted. A module on the bus may prevent the exclusive unmodified attribute by asserting **tf*** on read shared transactions.

If a cache observes a shareable read transaction for the cache line its processor has requested, it may snarf the data during which action it should assert **tf***. The resulting data is given to the processor, stored in the cache, and the shared unmodified attribute is set for the cache line.

8.1.3.3 Write miss process

When a processor issues a write, if the location requested is exclusive unmodified or exclusive modified, the operation is said to be a *write hit*, and no bus transaction is needed. If the location is not tagged exclusive, the operation is said to be a *write miss*, even if the location is valid. A write miss causes the cache to arbitrate for the bus. When the cache obtains the bus, there are two cases. If the cache contains a shared unmodified copy of the cache line, it issues an address only *invalidate* transaction. If the cache does not contain a shared unmodified copy of the cache line, it issues a *read modified* transaction. After the bus transaction, the data is taken from the processor, merged with the rest of the cache line, stored in the cache, and tagged exclusive modified.

While the cache is waiting to become master of the bus, it may snarf shareable transactions. This will allow the cache to use an address only invalidate transaction instead of a read modified transaction.

8.1.3.4 Copyback process

For a cache to service the reads and writes a processor generates, empty space in the cache is required to hold the data. After running for a while, the locations available to the cache to service a particular read or write miss may be in use. When this occurs, the cache will have to free up a space to service the cache miss. Locations that are not exclusive modified can be reused with no bus transaction. Locations that are exclusive modified require a bus transaction to copy a cache line from the cache back to memory to bring the system up to date before the cache location can be reused.

The copyback process can be used in other instances where restoring data to memory is needed. For example, a module may flush cache lines during context switching. Maintenance, diagnostic, and live withdrawal procedures will also use the copyback process to bring the system up-to-date.

The process starts when the cache has an exclusive modified line that it wants to change to invalid or shared unmodified. The cache arbitrates for the bus and, when it becomes master, initiates a copyback transaction

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

that copies the cache line back to memory. The master may keep a copy of the line changing it to shared unmodified if it asserts **tf*** during the connection phase.

While the cache is waiting to become master of the bus, another module may request a copy of the cache line. If this occurs, the module is to intervene and supply the data during the transaction. The cache is able at that time to cancel its copyback transaction.

8.1.3.5 I/O Controller optimization

DMA-based input/output controllers can read and write to cache coherent memory. The protocol requires that modules follow the cache coherence protocol. Modules are required to implement a one-line cache if they need to transfer partial cache lines. Two special transactions, *read invalid* and *write invalid*, are available to reduce the overhead of I/O buffer transfers.

The read invalid transaction is used by modules that do not have a valid cached copy before the transaction and will not have one after the transaction. The read invalid transaction is the same as the read shared transaction, except that if **TF*** is not asserted there will be no shared copies after the transaction.

The write invalid transaction is used by modules to write new data to all bytes in a cache line. Since none of the cache line is to remain the same, the module does not need the previous contents of the cached copy. The write invalid transaction invalidates all other copies of the cache line in the system and writes the new data to memory. At the successful completion of the transaction, no cache in the system contains a valid copy of the cache line.

Strictly speaking, cache coherence is not preserved by the write invalid transaction. Multiple simultaneous write invalid transactions and mixtures of write invalid transactions with other sources of modification will not be uniquely ordered in the system. It is assumed that reading and writing locations in a DMA buffer with a live DMA stream coming in is a programming error. The write invalid transaction is meant to be used to initialize a system data buffer. It is not meant to be used for ongoing cooperative updating of a shared memory data structure.

8.1.3.6 Examples

Figure 50 shows a model of a system that will be used in various cache coherence examples.

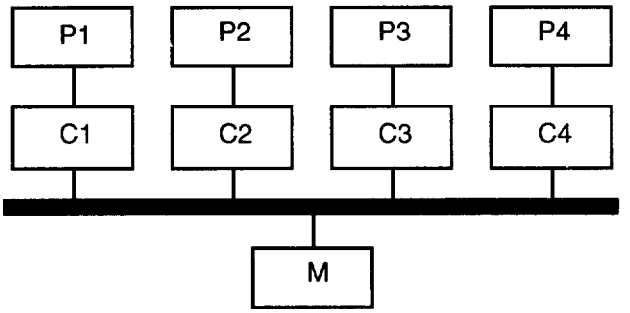


Figure 50—Connected cache coherence model

The following examples are not meant to show a typical sequence, but rather protocol details. A typical case would not have simultaneous transactions on the same address; each transaction would complete without interference. Usually, the system implementor would not have many processors updating the same cache line without synchronizing variables or other programming techniques to coordinate system level transactions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.1.3.6.1 One processor reads, then writes, a location

This example illustrates a read miss followed by a write miss. The example begins with all caches invalid. The action proceeds as follows:

1) Processor P1 reads a location from cache C1.
2) Cache C1 looks up the cache tag for the location and determines that the cache line is invalid.
3) Cache C1 arbitrates for the bus, becomes master, and then performs a read shared transaction.
4) When the transaction is complete, cache C1 determines that the transaction has completed without error and that **TF*** was not asserted.
5) Cache C1 sets the exclusive unmodified attribute for the cache line.
6) Cache C1 then provides the data to processor P1.
7) Processor P1 may then perform any number of reads, all of which will be read hits.
8) Processor P1 eventually writes to the exclusive unmodified cache line.
9) Cache C1 sets the coherence attribute to exclusive modified for the cache line.
10) Cache C1 then merges the write data into the cache line.
11) All subsequent writes by P1 to the cache line are write hits.

Eventually, cache C1 will return the data to the system by performing a copyback transaction or by intervening in a read transaction.

As shown in this example, one bus transaction is all that is required to satisfy any number of read requests from a single processor to an unshared cache line. Only two bus transactions are required to satisfy any number of writes from a single processor to an unshared cache line. These operations are statistically the most common in a shared memory system.

8.1.3.6.2 All Processors simultaneously read and write a location

This example illustrates four processors simultaneously reading a location and then writing the location. The example begins with all caches invalid. The action proceeds as follows:

1) All four processors attempt a read from their caches.
2) All four caches look up the cache tag for the location and determine it is invalid.
3) The four caches then arbitrate for the bus. Cache C1 wins the arbitration and becomes master.
4) Cache C1 then performs a read shared transaction.
5) The other three caches assert **tf*** and **bc*** to receive the data cache C1 is requesting.
6) When the transaction is complete, all of the caches determine that the transaction has completed without error and that **TF*** was asserted.
7) All the caches then mark the cache line shared unmodified and provide the data to their processors.
8) All four processors then simultaneously write to the cache line.
9) All four caches then arbitrate for the bus and C1 wins the arbitration and becomes master.
10) Cache C1 then performs an invalidate transaction.
11) When the three other caches snoop the invalidate transaction, they change the cache line to invalid.
12) When the transaction is complete and cache C1 determines that the transaction has completed without error it marks the line exclusive modified and merges the data from processor P1 with the cache line.
13) Caches C2–C4, after losing the arbitration in 9), again arbitrate for the bus and C2 wins and becomes master.
14) Cache C2 then performs a read modified transaction.
15) Cache C1 snoops the transaction, intervenes, marks the line invalid, and supplies the data instead of memory.
16) When the transaction is complete and cache C2 determines that the transaction has completed without error, it marks the line exclusive modified and merges the data from processor P2 with the cache line.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

17) Caches C3 and C4, after losing the arbitration in 13), again arbitrate for the bus and C3 wins and becomes master.
18) Cache C3 then performs a read modified transaction.
19) Cache C2 snoops the transaction, intervenes, marks the line invalid, and supplies the data instead of memory.
20) When the transaction is complete and cache C3 determines that the transaction has completed without error, it marks the line exclusive modified and merges the data from processor P3 with the cache line.
21) Cache C4, after losing the arbitration in 17), again arbitrates for the bus, wins, and becomes master.
22) Cache C3 snoops the transaction, intervenes, marks the line invalid, and supplies the data instead of memory.
23) Cache C4 then performs a read modified transaction.
24) When the transaction is complete and cache C4 determines that the transaction has completed without error, it marks the line exclusive modified and merges the data from processor P4 with the cache line.

The example has shown that it is possible to combine shareable accesses, but that each exclusive access requires an exclusive bus transaction.

8.1.3.6.3 Cache resident fetch and add

A cache line with the exclusive attribute can be used for atomic cache based read modify write sequences for synchronization, without involving the bus. These read modify write sequences may include the traditional test and set or exchange memory sequences, or the newer fetch and add or compare and swap sequences. All of these sequences can be guaranteed to be atomic if the cache line containing the synchronization variable has the exclusive attribute. These atomic sequences are local module implementation techniques and are not specified in this International Standard.

The following example illustrates how four processors simultaneously get a unique number from a synchronizing variable. The example begins with all caches invalid and the memory location zero. The action proceeds as follows:

1) All four processors attempt a read from their caches requesting exclusive ownership.
2) All four caches look up the cache tag for the location and determine it is invalid.
3) The four caches then arbitrate for the bus. Cache C1 wins the arbitration and becomes master.
4) Cache C1 then performs a read modified transaction.
5) When the transaction is complete and cache C1 determines that the transaction has completed without error, it marks the line exclusive modified and gives the datum from the memory location, which is 0, to processor P1. Processor P1 increments the datum.
6) Cache C1 then merges 1 into the cache line.
7) Caches C2–C4, after losing the arbitration in 3), again arbitrate for the bus and C2 wins and becomes master.
8) Cache C2 then performs a read modified transaction. Cache C1 intervenes, supplying the current data value, and marks the cache line invalid.
9) When the transaction is complete and cache C2 determines that the transaction has completed without error, it marks the line exclusive modified and gives the datum from the memory location, which is 1, to processor P2. Processor P2 increments the datum.
10) Cache C2 then merges 2 into the cache line.
11) Caches C3 and C4, after losing the arbitration in 7), again arbitrate for the bus and C3 wins and becomes master.
12) Cache C3 then performs a read modified transaction. Cache C2 intervenes, supplying the current data value, and marks the cache line invalid.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13) When the transaction is complete and cache C3 determines that the transaction has completed without error, it marks the line exclusive modified and gives the datum from the memory location, which is 2, to processor P3. Processor P3 increments the datum.
14) Cache C3 then merges 3 into the cache line.
15) Cache C4, after losing the arbitration in 11), again arbitrates for the bus, wins, and becomes master.
16) Cache C4 then performs a read modified transaction. Cache C3 intervenes, supplying the current data value, and marks the cache line invalid.
17) When the transaction is complete and cache C4 determines that the transaction has completed without error, it marks the line exclusive modified and gives the datum from the memory location, which is 3, to processor P4. Processor P4 increments the datum.
18) Cache C4 then merges 4 into the cache line.

As shown in this example it is possible to coordinate *n* processors with *n* bus transactions.

8.1.4 Cache coherence using split transactions on a single bus segment

Subclause 8.1.4 describes the use of split transactions in a cache coherent system with a single bus segment. Split transactions are used when the access time of a module is slow compared to bus access time.

All modules are not required to be capable of split transactions. Split transactions must be enabled by setting the split enable bit of the Logical Common Control register as described in 7.2. In general, split transactions will be enabled only if all modules in the system are capable of responding to split transactions. If a module is requested to split a transaction but is not capable it asserts **te***.

Split transactions are only used when needed. Each module that may want to split a transaction checks whether it needs to before causing a transaction to split.

8.1.4.1 Split transactions

Cache and memory modules determine whether they need to split by decoding the address and command of each cache coherent read transaction that they snoop. If the module is responsible for the address and can't respond immediately, it asserts **sr***.

A memory module may assert **sr*** to force split transactions. If the memory module does not observe **SW***, it shall generate a shared response to a read shared or read invalid transaction, and shall generate a modified response to a read modified command. The master may observe **IV*** and assert **sw***. If the memory module observes **SW***, it shall not generate any response transaction. A cache that asserts **iv*** may assert **sr*** to force split intervention. A cache that asserts **iv*** and detects **SR*** shall use split intervention or shall assert **te***.

8.1.4.1.1 Shared response

If a module splits a read shared or read invalid transaction, it must eventually respond with a shared response transaction.

A shared response transaction may be snarfed by any cache. The memory must snarf data from shared response transactions.

If no other cache signals that it has the cache line or that it is snarfing by asserting **tf*** on a shared response transaction, the line may be tagged exclusive unmodified in the requester. This allows the processor to update the cache line to exclusive modified without further bus transactions. Modules may prevent the exclusive unmodified state by asserting **tf*** on read shared and shared response transactions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.1.4.1.2 Modified response

If a module splits a read modified transaction, it must eventually respond with a modified response transaction.

The modified response transaction may not be snarfed. The module that originally generated a read modified or invalidate transaction and subsequently receives a modified response, changes its cache line state to exclusive modified.

8.1.4.2 Single outstanding transaction per cache line

Modules that split a transaction are required to assert **wt*** if any other module initiates a request transaction to the same cache line. This forces a limit of a single outstanding transaction per cache line. When a module initiating a transaction receives wait status, it is to wait until a shared response or modified response is snooped to the cache line and then it may repeat its request.

In the case of a shared response transaction, a module may snarf the data and satisfy its request without a bus transaction.

This protocol can be thought of as a queue of requests to a particular cache line. The module at the head of the queue received split status to its request and is guaranteed completion. All other modules wait until the completion of the modules first transaction. Then all waiting modules arbitrate for the bus. Arbitration picks one module as the next head of the queue. All of the losing modules snoop the winning module's transaction and go back to the waiting state without wasting transactions on the bus.

This protocol does not provide deterministic ordering of the queue. It depends on arbitration to give fair or, at worst, random ordering of the accesses. A pathological timing sequence may lead to the indefinite starvation of some module.

All exclusive transactions in the queue must be handled individually. While waiting, modules must invalidate when they snoop successful invalidating transactions.

8.1.4.3 Requester and responder attributes

Split capable modules use a set of responder attributes that they associate with cache lines for which they owe a response. Modules set responder shared if they split read shared or read invalid transactions. Modules set responder exclusive if they split read modified transactions.

Split capable modules use a set of requester attributes that they associate with cache lines for which they are owed a response. Modules set requester shared if they initiate a read shared or read invalid transaction which is split by another module. Modules set requester exclusive if they initiate a read modified transaction which is split by another module.

Modules set requester waiting if they receive wait status in a transaction. Modules also may set requester waiting if they snoop a transaction in which either **SR*** or **WT*** is asserted.

8.1.4.4 I/O controller optimization

As in the connected case, the write invalid transaction does not guarantee the unique system-wide ordering that is cache coherence.

A write invalid transaction cancels any requester or responder attributes that it encounters; any such modules must retry their transactions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A write invalid transaction is not shareable, and therefore not snarfable by any cache.

A read modified transaction followed by a copyback transaction is used to update partial cache lines in a cache coherent manner. If the read modified transaction is split, the data comes back in a modified response transaction. The master of the modified response transaction is different than the I/O controller module, so that other transactions may occur between the completion of the modified response transaction and the copyback transaction. The module must snoop matching addresses to ensure cache coherence until the copyback transaction is completed.

8.1.4.5 Examples

Figure 51 is used again to show a model of a system that will be used in the single bus segment split transaction cache coherence examples. The memory M splits all read transactions. The caches C1, C2, and C3 all use split intervention. The following examples are not meant to show a typical sequence, but rather protocol details.

8.1.4.5.1 One processor reads, then writes, a location

This example illustrates a read miss followed by a write miss. The example begins with all caches invalid. The action proceeds as follows:

1) Processor P1 reads a location from cache C1.
2) Cache C1 looks up the cache tag for the location and determines that the cache line is invalid.
3) Cache C1 arbitrates for the bus, becomes master, and then performs a read shared transaction.
4) The memory M splits the transaction and sets its responder shared attribute for the requested cache line.
5) When the transaction is complete, cache C1 determines that the transaction has been split and sets its requester shared attribute for the requested cache line.
6) When the data is available, the memory performs a shared response transaction.
7) When the transaction is complete, cache C1 and the memory M determine that the transaction has completed without error.
8) The memory M then clears its responder shared attribute.
9) The cache C1 then clears its requester shared attribute and, since no other cache asserted **tf***, changes the cache lines state to exclusive unmodified.
10) Cache C1 then provides the data to processor P1.
11) Processor P1 may then perform any number of reads, all of which will be read hits.
12) Processor P1 eventually writes to the exclusive unmodified cache line.
13) Cache C1 sets the cache line state to exclusive modified.
14) Cache C1 then merges the write data into the cache line.
15) All subsequent writes by P1 to the cache line are write hits.

As shown in this example the processors transaction is split into two bus transactions. The cache line state is updated at the completion of the sequence.

8.1.4.5.2 Three processors simultaneously read and write a location

This example illustrates three processors P1, P2, and P3 simultaneously reading a location and then writing the location. The example begins with all caches invalid. The action proceeds as follows:

1) The three processors attempt a read from their caches.
2) The three caches look up the cache tag for the location and determine it is invalid.
3) The three caches then arbitrate for the bus. Cache C1 wins the arbitration and becomes master.
4) Cache C1 then performs a read shared transaction.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5) Caches C2 and C3 assert **tf*** and **bc*** to receive the data cache C1 is requesting. The memory M asserts **sr*** to split the transaction.
6) When the transaction is complete, all of the caches determine that the transaction has been split and therefore no data was transferred.
7) The cache C1 sets its requester shared attribute, the memory M sets its responder shared attribute, and the other caches sets their requester waiting attributes.
8) When the data is available, the memory M performs a shared response transaction. Caches C2 and C3 assert **tf*** and **bc*** to receive the data memory M is supplying.
9) When the transaction is complete, the modules determine that the transaction has completed without error.
10) The memory M then clears its responder shared attribute.
11) The cache C1 then clears its requester shared attribute and changes the cache lines state to shared unmodified.
12) The other caches then clear their requester waiting attributes and change their cache line states to shared unmodified.
13) The caches then provide the data to their processors.
14) The three processors then simultaneously attempt a write to the cache line.
15) The three caches then arbitrate for the bus and C1 wins the arbitration and becomes master.
16) Cache C1 then performs an invalidate transaction.
17) Caches C2 and C3 snoop the invalidate transaction and changes their cache line to invalid.
18) When the transaction is complete and cache C1 determines that the transaction has completed without error, it marks the line exclusive modified and merges the data from processor P1 with the cache line.
19) Caches C2 and C3 then arbitrate for the bus and C2 wins and becomes master.
20) Cache C2 then performs a read modified transaction .
21) The memory M asserts **sr*** to split the transaction. Cache C1 snoops the transaction, asserts **sr*** to split the transaction, and asserts **iv*** to cause the cache C2 to assert **sw*** to notify the memory that it has no obligation to supply the requested data. Cache C2 sets its requester exclusive attribute. Cache C1 sets its responder exclusive attribute. Cache C3 maintains its requester waiting attribute.
22) When cache C1 has the data ready, it arbitrates for the bus and performs a modified response transaction. Cache C2 accepts the data.
23) Cache C2 sets its cache line state to exclusive modified. Cache C1 sets its cache line state to invalid. Cache C2 clears its requestor exclusive attribute. Cache C1 clears its responder exclusive attribute. Cache C3 clears its requestor waiting attribute.
24) Cache C3 arbitrates for the bus and performs a read modified transaction.
25) The memory M asserts sr* to split the transaction. Cache C2 snoops the transaction, asserts **sr*** to split the transaction, and asserts **iv*** to cause the cache C3 to assert **sw*** to notify the memory that it has no obligation to supply the requested data. Cache C3 sets its requester exclusive attribute. Cache C2 sets its responder exclusive attribute.
26) When cache C2 has the data ready, it arbitrates for the bus and performs a modified response transaction. Cache C3 accepts the data.
27) Cache C2 sets its cache line state to exclusive modified. Cache C1 sets its cache line state to invalid. Cache C2 clears its requestor exclusive attribute. Cache C1 clears its responder exclusive attribute.

The example has shown that it is possible to combine shareable accesses, but that each exclusive access requires an exclusive bus transaction.

8.1.5 Using split transactions to delay invalidation completions

Large systems may be built by connecting a bus segment to other modules with bus bridges. These bus bridges will have slow access to data on the remote bus segments, so they use split transactions to access data as previously described. In addition, bus bridges have slow access to cache tags on the remote bus segments. This clause describes the use of split transactions to guarantee that all remote caches are invalid.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The split transaction shared memory architecture, shown in figure 51, consists of the shared memory architecture extended with *cache agent* modules and *memory agent* modules.

The figure is intended to show that the cache agents, memory agents, and associated modules are connected with some technology that may well be independent of this specification. The figure is not intended to show packaging into boards or boxes, or other implementation details. Some implementations may put both cache agent and memory agent modules together to implement a cache coherent bus bridge. Some implementations may use hierarchical caching with multiple levels of cache coherent protocols on multiple ISO/IEC 10857 bus segments. Some implementations may use cache coherent bridges to other bus standards.

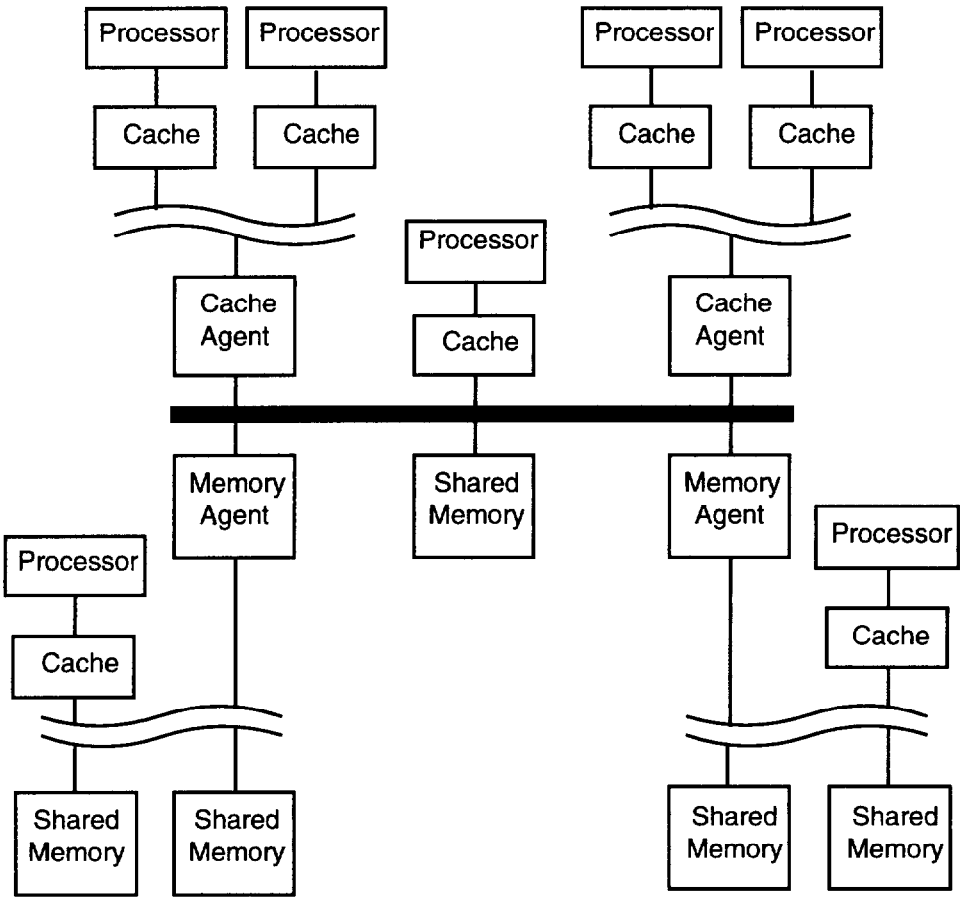


Figure 51—Split response architecture

8.1.5.1 Cache agents

A module is said to be a cache agent if it uses split transactions to assume all the rights and responsibilities of some number of remote cache modules. In a hierarchical cache design, the cache agent assumes the rights and responsibilities of cache modules further from memory in the hierarchy.

Cache agents must maintain tags containing the cache line state for each cache line contained in any of the caches for which it is responsible. Nothing in this specification implies that a cache agent must keep a copy of the data that goes through it or is kept in caches or cache agents connected through it. A cache agent may have a data segment in its cache.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A cache agent splits a transaction if it has the modified attribute for the cache line, and needs to intervene to transfer the up to date copy from a remote cache module.

A cache agent splits a transaction when it has the valid attribute for the cache line, and cannot immediately invalidate the remote cache modules for which it is responsible. A cache agent also asserts **tf***, indicating that some caches further away from memory in a hierarchy may still be valid.

The transaction that has arbitrated for and won the bus segment will have priority in the cache agent over a conflicting transaction to the same address coming from any remote caches.

8.1.5.2 Memory agents

A module is said to be a memory agent if it uses split transactions to assume all the rights and responsibilities of some number of remote memory modules. In a hierarchical cache design, the memory agent also assumes the rights and responsibilities of memory modules closer to memory in the hierarchy.

Every memory location that is accessible by the cache coherent protocols must have exactly one selected module that services accesses to its location on each bus segment.

A memory agent must be able to get the data from an exclusive modified cache line when a remote cache requests it. If the cache line is exclusive modified, the memory agent must use a read request to get the data. If the cache line is not exclusive modified, any memory agent read request will fail with no intervener. If a cache line is exclusive unmodified, it may change to any other cache line state without a bus-visible transaction. Memory agents keep track of each cache line which it is responsible for that has the exclusive modified state in the hierarchy above it by setting the memory line modified attribute for that line. The memory agent must prevent the exclusive unmodified cache line state from occurring so that it can monitor all state changes to exclusive modified.

Nothing in this specification implies that a memory agent must keep a copy of data that goes through it or is kept in caches or cache agents connected through it. A memory agent may have a data segment in its cache to increase the speed of transactions.

The memory agent splits a transaction to assign the read and write permissions of the locations it is handling for the bus segment that it is on. Write permission is requested by read modified or invalidate. Write permission is granted by a successful completion of the transaction, and any associated split response transaction. When the bus segment has read and write permissions, all caches and cache agents must be in the invalid, shared unmodified, and exclusive modified states. Read permission is requested by read shared, read modified, copyback with **tf*** asserted, or read invalid with **tf*** asserted. Read permission is granted by a successful completion of the transaction, and any associated split response transaction. When the bus segment has only read permission, all caches and cache agents must be in the invalid or shared unmodified states.

A memory agent must pass external invalidations to the bus segment if the bus segment has read permission. A memory agent must pass external interventions to the bus segment if the bus segment has write permission.

8.1.5.3 Split transactions

Cache agent and memory agent modules determine whether they need to delay their invalid acknowledgement by decoding the address and command of each cache coherent transaction that they snoop. If the module is responsible for the address and can't respond immediately, it asserts **sr*** or **tf*** as described below.

If more than one module splits a particular transaction, all of the modules that split the transaction split any response transactions they snoop until they complete their response. All of these modules eventually issue a response transaction or allow a snooped response transaction to complete without further splitting.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The following cache coherent transactions can have delayed invalidation acknowledgements: read modified, invalidate, write invalid, and modified response.

8.1.5.3.1 Read modified

Any module can delay the invalidation acknowledgement by asserting **tf*** to a read modified transaction.

If **SR*** is also asserted by the module that will produce the data copy, then the read modified is split for both the access and the invalidation acknowledgements. In this case, each asserter of **tf*** must eventually issue a response transaction or allow a snooped response transaction to complete without further splitting.

If **SR*** is not asserted but **TF*** is asserted, the read modified did produce a valid connected data copy, but must be followed by a invalidate transaction to get a guaranteed invalidation acknowledgement. In this case, each asserter of **tf*** has no obligation until the invalidate transaction occurs.

8.1.5.3.2 Invalidate

Any module can delay the invalidation acknowledgement by asserting **sr*** to a invalidate transaction.

8.1.5.3.3 Write invalid

As in the connected case, the write invalid transaction does not guarantee the unique system-wide ordering that is cache coherence. However, all previously valid caches do have to indicate invalidation complete, so that the DMA controller can wait for all of the DMA'ed data to be observable before it posts its status block.

Any module can delay the invalidation acknowledgement by asserting **sr*** to a write invalid transaction.

8.1.5.3.4 Modified response

If a module splits a read modified, invalidate, or write invalid transaction, it must eventually respond with a modified response transaction or allow another modules modified response to complete without further splitting.

Cache agents or memory agents may split with **sr*** a modified response for a cache line that is not yet known to be invalid. Cache agents that split a modified response also assert **tf***.

The modified response transaction may not be snarfed. The module that originally generated a read modified or invalidate transaction and subsequently receives a modified response without **SR*** changes its cache line state to exclusive modified.

8.1.5.4 Requester and responder attributes

Cache agents and memory agents use requester and responder attributes as previously described. In addition, modules set responder invalidate if they split invalidate transactions. Modules set responder write invalid if they split write invalid transactions. Modules set requester exclusive if they initiate an invalidate transaction that is split by another module. Modules set requester write invalid if they initiate a write invalid transaction that is split by another module.

8.1.5.5 Multiple outstanding transactions per cache line

A system may consist of several bus segments, all cooperatively implementing one shared memory space, with cache coherence implemented with hardware protocols using bus bridge modules with one or more cache agents and memory agents. In such a system, the bus segments run with independent timing. Even

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

though the protocols only allow one outstanding transaction per cache line on each bus segment, there may be multiple simultaneous outstanding transactions per cache line in the total system.

Using hierarchical caches with the inclusion principle causes all possibly conflicting, simultaneous transactions to collide. A system of hierarchical caches exhibits the inclusion principle if a single path exists for each cache line from a cache to memory and each cache agent contains the sum of the attributes for all cache lines it is responsible for. In all the examples shown, the inclusion principle is met.

Each transaction can be considered to be coming from a cache or cache agent, headed towards memory, or from a memory or memory agent, headed towards caches.

Some transaction types may be passed by a cache agent away from memory. Such transactions are as follows:

1) A read shared or read modified transaction to read a modified, up-to-date copy.
2) An invalidate transaction to force all remote caches to be invalid.
3) A write invalid transaction to force all remote caches to be invalid and all pending transactions to be retried.

Any transaction type may be passed by a memory agent towards memory. All transactions heading towards memory imply that the bus segment behind them no longer has the exclusive attribute. Transactions going towards memory may fork off transactions going away from memory to ensure that all remote caches are invalid.

Transactions from multiple caches or cache agents to one particular bus segment are ordered by arbitration and waiting protocols. Transactions going in the same direction are handled in order and can never collide. Transactions going in the opposite direction can collide inside of bus bridges resulting in an *invalidate collision*. These collisions are always resolved with the transaction headed away from memory being handled first, and the transaction headed towards memory waiting until it observes the winner's completion.

Read transactions heading away from memory always pursue the exclusive attribute. If they collide with a transaction heading towards memory, that transaction must have the up-to-date copy in the transaction. This completes a read shared transaction. A read modified transaction gets the data copy, but has to continue away from memory as an invalidate transaction.

An invalidate transaction heading away from memory may encounter a bus segment with caches and or cache agents that are in the process of handling a split transaction. The memory agent issues the invalidate transaction, without asserting **wt***. The caches all invalidate without disturbing their split transaction state. The cache agents make sure that their remote caches are being invalidated, asserting **sr*** if some remote caches are not yet invalidated. The cache agents report completion with a modified response transaction. All cache agents snoop, and assert **sr*** and **tf*** if not yet completely invalidated. The memory agent snoops, and asserts **sr*** to prevent the original requestor from completing. When the memory agent observes a modified response transaction without **TF***, it knows that all caches connected through the bus segment are guaranteed to be invalid.

A write invalid transaction heading away from memory may encounter a bus segment with caches and or cache agents that are in the process of handling a split transaction. The memory agent issues the write invalid transaction as an address only cycle, without asserting **wt***. The caches all invalidate, cancelling any pending split transaction state. The cache agents make sure that their remote caches are being invalidated by passing the write invalid on, asserting **sr*** if some remote caches are not yet invalidated. The cache agents report completion with a modified response transaction. All cache agents snoop, and assert **sr*** if not yet completely invalidated. The memory agent snoops. When the memory agent observes a modified response transaction without **sr***, it knows that all caches connected through the bus segment are guaranteed to be

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

invalid. After the modified response transaction without **sr***, any caches with cancelled split transactions are allowed to retry.

The write invalid transaction may be split by any number of bus bridges. If it splits, the bridge must eventually respond with a modified response transaction, or allow another bridge's modified response transaction to complete without further splitting.

A cache agent or memory agent that performs a delayed invalidation asserts **sr*** to the invalidate command, and is later responsible for generating a modified response transaction or allowing another agent's modified response transaction to complete without asserting **sr***.

A cache agent or memory agent that has the up-to-date copy of the cache line asserts **sr*** to a read modified command. It is later responsible for generating a modified response transaction with the up-to-date copy.

A cache agent that performs a delayed invalidation asserts **tf*** to the read modified command. If a memory agent asserts **sr***, then the cache agent performs a split invalidation as above. If no memory agent asserts **sr***, then the master must follow the read modified transaction with an address only invalidate transaction, allowing the cache agent to split by asserting **sr***. If the invalidate transaction immediately follows in the same tenure, then no opportunity for live lock is introduced.

8.1.5.6 Examples

Figure 52 shows one cache line in module M, shared by the ten processors through the ten caches, four cache agents, four memory agents, and five bus segments. For the sake of the example, all modules are given hierarchical numbers. All bus segments take the label of their supporting memory or memory agent.

The example topology should be big enough to show every split response shared memory protocol failure case. It has multiple copies of an entity whenever it has an entity. It has leaf, middle, and root bus segments.

The example is not meant to specify implementation details. For example, the cache agent and memory agent end of the bus bridges are drawn with point to point links; many other topologies are possible.

The examples are not meant to show a typical sequence, but rather protocol details. A typical case would not have simultaneous transactions on the same address; each transaction would complete without interference. Usually, the system implementor would not have many processors updating the same cache line without synchronizing variables or other programming techniques to coordinate system level transactions.

8.1.5.6.1 One processor reads and writes a location

This example illustrates a read miss followed by a write miss. The example begins with all caches invalid. The action proceeds as follows:

1) Processor P411 reads a location from cache C411. Cache C411 looks up the cache tag for the location and determines that the cache line is invalid.
2) Cache C411 arbitrates for the bus MA41, becomes master, and then performs a read shared transaction.
3) The memory agent MA41 splits the transaction, sets its responder shared attribute for the requested cache line, and passes the read request to cache agent CA41 with an unspecified protocol.
4) When the transaction is complete, cache C411 determines that the transaction has been split and sets its requester shared attribute for the requested cache line.
5) Cache agent CA41 then arbitrates for bus MA4, becomes master, and then performs a read shared transaction.
6) The memory agent MA4 splits the transaction, sets its responder shared attribute for the requested cache line, and passes the read request to cache agent CA4 with an unspecified protocol.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

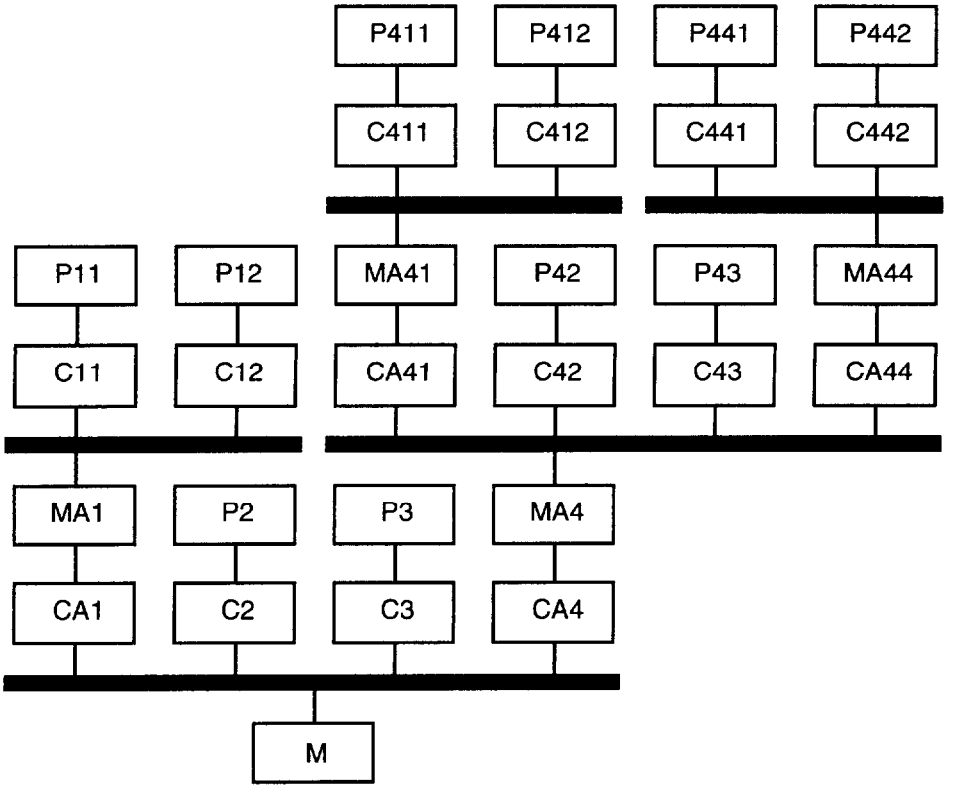


Figure 52—Multiple bus segment split transaction example model

7) When the transaction is complete, cache agent CA41 determines that the transaction has been split and sets its requester shared attribute for the requested cache line.
8) Cache agent CA4 then arbitrates for bus M, becomes master, and then performs a read shared transaction. Memory M splits by asserting **sr***. Cache agent CA4 sets requester shared. Memory M sets responder shared.
9) The memory M obtains the data, arbitrates for the bus, and performs a shared response transaction. Memory M supplies the data. Cache agent CA4 accepts the data, changes the state of the cache line to exclusive unmodified, and passes the data to MA4. Cache agent CA4 resets requester shared. Memory M resets responder shared.
10) Memory agent MA4 then arbitrates for bus MA4 and performs a shared response transaction asserting **tf*** to prevent the exclusive unmodified state. When the transaction completes successfully, it clears its responder shared attribute.
11) Cache agent CA41 accepts the data, changes the state of the cache line to shared unmodified, clears its requester shared attribute, and passes the data to memory agent MA41.
12) Memory agent MA41 then arbitrates for bus MA41 and performs a shared response transaction asserting **tf*** to prevent the exclusive unmodified state. When the transaction completes successfully, it clears its responder shared attribute.
13) Cache C411 accepts the data, changes the state of the cache line to shared unmodified, clears its requester shared attribute, and passes the data to processor P411.
14) This satisfies the read initiated by processor P411. Processor P411 may then perform any number of reads, all of which will be read hits.
15) Processor P411 eventually writes to the cache line. Cache C411 looks up the cache tag for the location and determines that the cache line is not exclusive.
16) Cache C411 arbitrates for bus MA41, becomes master, and then performs an invalidate transaction.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17) The memory agent MA41 splits the transaction, sets its responder invalidate attribute for the requested cache line, and passes the invalidate request to cache agent CA41.
18) When the transaction is complete, cache C411 determines that the transaction has been split and sets its requester exclusive attribute for the requested cache line.
19) Cache agent CA41 then arbitrates for bus MA4, becomes master, and then performs an invalidate transaction.
20) The memory agent MA4 splits the transaction, sets its responder invalidate attribute for the requested cache line, and passes the invalidate request to cache agent CA4.
21) When the transaction is complete, cache agent CA41 determines that the transaction has been split and sets its requester exclusive attribute for the requested cache line.
22) Cache agent CA4 changes its cache line state from exclusive unmodified to exclusive modified and passes the modified response to MA4. It should be noted that with an integrated implementation of bus bridge 4, MA4 may have had access to the tags in CA4 and would not have needed to split the transaction in 19).
23) Memory agent MA4 then arbitrates for bus MA4 and performs a modified response transaction. When the transaction completes successfully, it clears its responder invalidate attribute.
24) Cache agent CA41 accepts the modified response, changes the state of the cache line to exclusive modified, clears its requester exclusive attribute, and passes the response to MA41.
25) Memory agent MA41 then arbitrates for bus MA41 and performs a modified response transaction. When the transaction completes successfully it clears its responder invalidate attribute.
26) Cache C411 accepts the modified response, changes the state of the cache line to exclusive modified, clears its requester exclusive attribute, and merges the data from processor P411 with the cache line and stores it in the cache.

All subsequent reads and writes from P411 are hits. The example has shown how transactions proceed through multiple bus bridges, and how cache line attributes are accumulated using the inclusion principle.

8.1.5.6.2 Four processors simultaneously read and write a location

This example illustrates four processors P11, P2, P411, and P412 simultaneously reading a location and then writing the location. The example begins with all caches invalid. The original value in memory is "0123". The four processors write "A---", "-B--", "--C-", and "---D", respectively, where only the part of the cache line containing the letter is written. The action proceeds as follows:

1) All four processors attempt a read from their caches.
2) The four caches look up the cache tag for the location and determine it is invalid.
3) The four caches then arbitrate for their respective bus. Cache C411 wins the arbitration and becomes master of bus MA41. Cache C411 then performs a read shared transaction which memory agent MA41 splits. Cache C411 sets its requester shared attribute. Memory agent MA41 sets its responder shared attribute and passes the read request to cache agent CA41. As a result of snooping the transaction on the bus, cache C412 sets its requester waiting attribute.
4) Cache agent CA41 arbitrates for bus MA4, becomes master, and performs a read shared transaction which memory agent MA4 splits. Cache Agent CA41 sets its requester shared attribute. Memory agent MA4 sets its responder shared attribute and passes the read request to CA4.
5) Meanwhile, C11 wins the arbitration and becomes master of bus MA1. Cache C11 then performs a read shared transaction which memory agent MA1 splits. Cache C11 sets its requester shared attribute. MA1 sets its responder shared attribute and passes a read request to cache agent CA1.
6) Cache agent CA1, cache C2, and cache agent CA4 all arbitrate for bus M. Cache agent CA4 wins the arbitration and becomes master of bus M. Cache agent CA4 then performs a read shared transaction and the memory M splits by asserting **sr***. Cache agent CA4 sets requester shared. Memory M sets responder shared. Cache agent CA1 and C2 set requester wait.
7) Memory M obtains the data, arbitrates for the bus, performs a shared response transaction, and supplies the data "0123". Cache agent CA1 cache C2, and cache agent CA4 all snarf the data and clear

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

their requester attributes. Memory M clears its responder attribute. Cache agent CA4 changes the state of the cache line to shared unmodified and passes the read response to MA4.

8) Cache agent CA1, after snarfing the data, changes the state of the cache line to shared unmodified and passes the read response to MA1.
9) Cache C2, after snarfing the data and changing the state of the cache line to shared unmodified, has completed its read request and passes the data to processor P2.
10) Memory agent MA1 arbitrates for bus MA1, becomes master, and performs a shared response transaction transferring the data to cache C11. Memory agent MA1 clears its responder shared attribute. Cache C11 clears its requester shared attribute, changes the state of the cache line to shared unmodified, and provides the data to processor P11.
11) Memory agent MA4 arbitrates for bus MA4, becomes master, and performs a shared response transaction transferring the data to cache agent CA41. Memory agent MA4 then clears its responder shared attribute. Cache agent CA41 passes the data to MA41 and clears its requester shared attribute. CA41 changes the state of the cache line to shared unmodified.
12) Memory agent MA41 arbitrates for bus MA41, becomes master, and performs a shared response transaction transferring the data to cache C411. Cache C412 snarfs the data. Both caches change the state of the cache line to shared unmodified. Cache C411 clears its requester shared attribute. Cache C412 clears its requester waiting attribute.

This satisfies the read initiated by the four processors. The processors may then perform any number of reads, all of which will be read hits. The four caches and cache agents CA1, CA4, and CA41 all have the line marked shared unmodified.

13) All four processors then write to their caches.
14) The four caches look up the cache tag for the location and determine it is shared unmodified and they must invalidate copies in all other copies in the system to change the state to exclusive modified.
15) The four caches then arbitrate for their respective bus. Cache C2 wins the arbitration, becomes master of bus M, and performs an invalidate transaction which cache agents CA1 and CA4 split. Cache C2 sets its requester exclusive attribute and cache agents CA1 and CA4 set their responder invalidate attributes and pass the invalidate to their respective memory agents.
16) Memory agent MA1 arbitrates for bus MA1 and happens to win arbitration over cache C11. When memory agent MA1 becomes master of the bus, it performs an invalidate transaction causing cache C11 to change the state of the cache line to invalid. Memory agent MA1 passes the invalidate completion to CA1.
17) Cache C11 continues arbitration for bus MA1, becomes master, and performs a read modified transaction which memory agent MA1 splits. Cache C11 sets its requester exclusive attribute and memory agent MA1 sets its responder exclusive attribute and passes the read modified request to cache agent CA1.
18) Cache C411 then wins the arbitration for bus MA41, becomes master, and performs an invalidate transaction which MA41 splits. Cache C411 sets its requester exclusive attribute and memory agent MA41 sets its responder invalidate attribute and passes the invalidate request to CA41. Cache C412 snooped the transaction and set its requester waiting attribute thereby going into a waiting state without wasting a bus access.

At this point, all four of the caches are waiting for responses.

19) Cache agent CA1 becomes master of bus M and performs a modified response transaction. Cache agent CA4 splits the transaction with **sr*** and **tf*** because it has not completed its invalidate. Cache agent CA1 clears its responder invalidate attribute.
20) Cache agent CA1 then receives the read modified from memory agent MA1 and, since it is still master of bus M, performs a read modified transaction. Cache agent CA4 asserts **wt*** and cache agent CA1 sets its requester waiting attribute.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

21) Cache agent CA41 becomes master of bus MA4 and performs an invalidate transaction which MA4 splits. Cache agent CA41 sets its requester exclusive attribute and memory agent MA4 sets its responder invalidate attribute.

At this point, both cache agent CA4 and memory agent MA4 have their responder invalidate attribute set. This means that an *invalidate collision* has occurred in bus bridge CA4/MA4. The invalidate coming from cache C2 is processed first, and the invalidate coming from cache agent CA41 is handled second.

22) Memory agent MA4 clears its responder invalidate attribute and sets its responder exclusive attribute.
23) Memory agent MA4 then becomes master of bus MA4 and performs an invalidate transaction which cache agent CA41 splits. Memory agent MA4 sets its requester exclusive attribute. Cache agent CA41 sets its responder invalidate attribute and forwards the request to memory agent MA41.
24) Memory agent MA41 then becomes master of bus MA41 and performs an invalidate transaction which causes cache C411 to change the state of the cache line to invalid. Memory agent MA41 passes the invalidate completion to CA41.
25) Cache agent CA41 becomes master of bus MA4 and performs a modified response transaction. Memory agent MA4 passes the modified response to cache agent CA4 and clears its requester exclusive attribute.
26) Cache agent CA4 becomes master of bus M and performs a modified response transaction and clears its responder invalidate attribute. Cache C2 is allowed to change its cache line state to exclusive modified and merge the write data from processor P2. The value of the cache line is then "0B23". Cache agent CA1 snoops the modified response and is then able to clear its requester waiting attribute.

At this point, processor P2 has completed its write. The cache line has the exclusive modified attribute in cache C2 and is invalid in all other caches in the system.

27) Cache agent CA4 becomes master of bus M and performs a read modified transaction. Memory M splits by asserting **sr***. Cache C2 intervenes in the transaction by asserting **sr*** and **iv***. Cache agent CA4 asserts **sw***. Cache C2 sets responder exclusive. Cache agent CA4 sets requester exclusive. Cache agent CA1 sets its requester wait attribute.
28) Cache C2 obtains the data, arbitrates for the bus, and performs a modified response transaction. Cache C2 supplies the data "0B23" to cache agent CA4 and to the memory M. Cache C2 changes the state of the cache line to invalid and resets its responder exclusive attribute. Cache agent CA4 changes the state of the cache line to exclusive modified, resets its requester exclusive attribute, and passes the data "0B23" to memory agent MA4. Cache agent CA1 resets its requester wait attribute.
29) Cache agent CA1 then becomes master of bus M and performs a read modified transaction which cache agent CA4 splits with **sr*** and **iv***. Cache agent CA1 sets its requester exclusive attribute and cache agent CA4 sets its responder exclusive attribute.
30) Memory agent MA4 becomes master of bus MA4 and performs a modified response transaction supplying the data "0B23" to cache agent CA41. Memory agent MA4 clears its responder exclusive attribute and sets its memory line modified attribute. Cache agent CA41 clears its requester exclusive attribute and changes the state of the cache line to exclusive modified.
31) Since memory agent MA4 is master of bus MA4 and its requester exclusive attribute is set, it immediately performs a read modified transaction. Cache agent CA41 splits with **sr*** and **iv***. Memory agent MA4 sets its requester exclusive attribute and cache agent CA41 sets its responder exclusive attribute. Cache agent CA41 passes the read modified request on to memory agent MA41.
32) Memory agent MA41 becomes master of bus MA41 and performs a modified response transaction supplying the data "0B23" to cache C411. Memory agent MA41 clears its responder exclusive attribute and sets its memory line modified attribute. Cache C411 changes the state of its cache line to exclusive modified, clears its requester exclusive attribute, and merges the write data from processor P411. The value of the cache line is then "0BC3". Cache C412 clears its requester waiting attribute.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

33) Cache C412 becomes master of bus MA41 and performs a read modified transaction. Cache C411 intervenes in the transaction with **sr*** and **iv***. Cache C411 sets its responder exclusive attribute. Cache C412 sets its requester exclusive attribute. Memory agent MA41 sets its requester wait attribute.
34) Cache C411 obtains the data, arbitrates for the bus, and performs a modified response transaction. Cache C411 supplies the data "0BC3" to cache C412. Cache C412 merges the write data changing its value to "0BCD" and changes the cache line state to exclusive modified. Cache C411 changes the cache line state to invalid. Cache C411 resets its responder exclusive attribute. Cache C412 resets its requester exclusive attribute. Memory agent MA41 resets its requester wait attribute.
35) Memory agent MA41 becomes master of bus MA41 and performs a read modified transaction. Cache C412 intervenes in the transaction with sr* and iv*. Cache C411 sets its responder exclusive attribute. Memory agent MA41 sets its requester exclusive attribute.
36) Cache C412 obtains the data, arbitrates for the bus, and performs a modified response transaction. Cache C412 supplies the data "0BCD", clears its responder exclusive attribute, and changes its cache line state to invalid. Memory agent MA41 passes the data on to CA41 and clears its memory line modified attribute and requester exclusive attribute.
37) Cache agent CA41 becomes master of bus MA4 and performs a modified response transaction passing the data on to memory agent MA4. Cache agent CA41 clears its responder exclusive attribute and changes its cache line state to invalid. Memory agent MA4 passes the data on to cache agent CA4 and clears its memory line modified attribute and its requester exclusive attribute.
38) Cache agent CA4 becomes the master of bus M and performs a modified response transaction passing the data on to cache agent CA1. Cache agent CA4 clears its responder exclusive attribute and changes its cache line state to invalid. Cache agent CA1 clears its requester exclusive attribute, changes its cache line state to exclusive modified, and passes the data on to memory agent MA1.
39) Memory agent MA1 becomes the master of bus MA1 and performs a modified response transaction passing the data "0BCD" to cache C11. Memory agent MA1 clears its responder exclusive attribute and sets its memory line modified attribute. Cache C11 clears its requester exclusive attribute, changes its cache line state to exclusive modified, and merges the write data. The value of the cache line is finally "ABCD".

In general, the number of transactions expected in a split response environment is (number of requests) times (number of hierarchical levels) times two (request then response). The number of transactions can be greater if many transactions have to go to memory and then further to an exclusive modified cache; the number of transactions can be less if many transactions have locality in either space or time that allows the diameter of transaction sequences to be small or multiple transactions to be merged. Performance will strongly depend on the clustering of the data accesses in the physical topology of the system.

In the above example, the system handled eight requests for service (four reads and four writes) in 32 transactions. This example in the three-level hierarchy could easily have had 8*3*2 or 48 transactions. Reference locality and read merging were reasonably successful in reducing the number of transactions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.1.6 Summary of cache coherence commands and status:

	SR*	IV*	TF*	
Read shared	0	0	0	Con M, go to EU
	0	0	1	Con M, go to SU
	0	1	0	Con C, go to EU, M must snarf
	0	1	1	Con C, go to SU, M must snarf
	1	0	X	Spl M
	1	1	X	Spl C
Read modified	0	0	0	Con M, go to EM
	0	0	1	Con M, go to SU, then invalidate
	0	1	0	Con C, go to EM
	1	0	0	Spl M
	1	0	1	Spl M, Spl I
	1	1	0	Spl C
	X	1	1	illegal
Invalidate	0	0	0	go to EM
	1	0	0	Spl I
	X	X	1	illegal
	X	1	X	illegal
Copyback	0	0	0	all I
	0	0	1	some SU
	X	1	X	illegal
	1	X	X	illegal
Read invalid	0	0	0	Con M, all I
	0	0	1	Con M, some SU
	0	1	0	Con C, all I, M must snarf
	0	1	1	Con C, some SU, M must snarf
	1	0	X	Spl M
	1	1	X	Spl C
Write invalid	0	0	0	all I
	1	0	0	Spl I
	X	1	X	illegal
	X	X	1	illegal
Shared response	0	0	0	go to EU, M must snarf
	0	0	1	go to SU, M must snarf
	1	X	X	illegal
	X	1	X	illegal
Modified response	0	0	0	go to EM
	1	0	0	Spl I, all CA I
	1	0	1	Spl I, some CA not I
	0	0	1	illegal
	X	1	X	illegal

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Where the abbreviations mean the following:

"X"	—	don't care.
"Con M"	—	connected response from the memory.
"Con C"	—	connected response from an intervening cache.
"Spl M"	—	a split response has been committed by the memory.
"Spl C"	—	a split response has been committed by an intervening cache.
"Spl I"	—	a split response has been committed by at least one delayed invalidator.
"go to SU"	—	the requester goes to cache line state shared unmodified; some other cache may also be in cache line state shared unmodified.
"go to EU"	—	the requester goes to cache line state exclusive unmodified; every other cache is guaranteed to be in cache line state invalid.
"go to EM"	—	the requester goes to cache line state exclusive modified; every other cache is guaranteed to be in cache line state invalid.
"M must snarf"	—	memory must must update itself with the data (unless memory is the master of a split response).
"then invalidate"	—	the master must follow the read modified transaction with an invalidate transaction.
"all I"	—	every cache is guaranteed to be in cache line state invalid.
"some SU"	—	some cache may be in cache line state shared unmodified.
"all CA I"	—	every cache and cache agent is guaranteed to be in cache line state invalid; however, the memory agent is not in cache line state invalid.
"some CA not I"	—	some cache or cache agent is not in cache line state invalid.

8.1.7 Illegal attribute combinations

If the exclusive attribute is set for a cache line in one cache, then no other cache may have that line valid. This leads to a number of easily detectable illegal command sequences. Any of these conditions indicate that the requirements of cache coherence have been compromised and should result in the assertion of **TE***.

8.1.7.1 Shared unmodified

If a cache line is shared unmodified, and the cache observes a copyback transaction or a read transaction with **IV*** asserted, then another cache illegally has the exclusive modified attribute.

8.1.7.2 Exclusive unmodified

If a cache line is exclusive unmodified, and the cache observes a copyback transaction, a read transaction with **IV*** asserted, a read modified transaction with **TF*** asserted but **tf*** not asserted, or an invalidate transaction, then another cache illegally has the valid attribute.

8.1.7.3 Exclusive modified

If a cache line is exclusive modified, and the cache snoops a copyback transaction, a read modified transaction with **TF*** asserted, or an invalidate transaction, then another cache illegally has the valid attribute.

8.1.7.4 Memory line modified

If a memory agent cache line is memory line modified, and the memory agent observes a read transaction without **IV*** asserted, then a cache has illegally dropped the exclusive modified attribute. If memory agent cache line is memory line modified, and the memory agent observes an invalidate transaction, then a cache illegally has the shared unmodified attribute

If a memory agent cache line is not memory line modified, and the memory agent observes a read transaction with **IV*** asserted or a copyback transaction, then a cache illegally has the exclusive modified attribute.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2 Specification

The equations in this clause assume that transactions complete successfully. If **BS*** is asserted the command shall have no effect. If **TE*** is asserted the command shall have no effect or the error shall be reported to a higher level.

8.2.1 Module attributes

CACHE. A CACHE module shall maintain storage for at least one cache line. A CACHE shall maintain the attributes specified in 8.2.3 for each cache line.

CACHE_AGENT. A CACHE_AGENT is a CACHE module for which SPLIT_ENABLE & SPLIT_ACCEPTOR is true.

MEMORY. A MEMORY module shall be the ultimate source and/or destination of data which may be stored within it.

MEMORY_AGENT. A MEMORY_AGENT is a module for which SPLIT_ENABLE & SPLIT_ACCEPTOR is true, acting on a local bus in place of remote MEMORY.

8.2.2 Status attributes

CACHE_OP_CMPLT. Modules shall set CACHE_OP_CMPLT if TRANSACTION_COMPLETE & ~BUSY & ~TRANSACTION_ERROR | MULTIPLE_PACKET_MODE & ATTRIBUTES_QUEUED.

ATTRIBUTES_QUEUED. Modules shall set ATTRIBUTES_QUEUED if all the attributes necessary to perform a queued cache line operation have been stored such that the attributes can be later applied to the appropriate packet.

CACHED. A module shall set CACHED if ADDR_DECODED and the address is in the range set by the system implementor to cache coherent space and the module is capable of responding to cache coherent transactions and the module has a tag match for the cache line associated with the address. Modules shall maintain CACHED set until CACHE_OP_CMPLT.

SNARF_DATA. A module may set SNARF_DATA if CACHED & ((READ_SHARED | READ_INVALID) & SLAVE_WRITE | SHARED_RESPONSE) and shall maintain SNARF_DATA set until CACHE_OP_CMPLT. A memory or memory agent shall set SNARF_DATA if CACHED & (READ_SHARED | READ_INVALID | SHARED_RESPONSE) & SLAVE_WRITE and shall maintain SNARF_DATA set until CACHE_OP_CMPLT.

PREVENT_EU. A module may set PREVENT_EU if (MASTER | CACHED) & (READ_SHARED | SHARED_RESPONSE) and shall maintain PREVENT_EU set until CACHE_OP_CMPLT.

KEEP_COPY. A module shall set KEEP_COPY if (CACHED & ~INVALID_STATUS & (READ_INVALID | READ_SHARED | MASTER & (COPY_BACK | SHARED_RESPONSE)) & the module keeps ~INVALID_STATUS at the end of the transaction) and shall maintain KEEP_COPY set until CACHE_OP_CMPLT.

ASSERT_CS. A module shall set ASSERT_CS if (MASTER | CACHED) & (SNARF_DATA | KEEP_COPY & ~READ_INVALID | PREVENT_EU) | (CACHE_AGENT & (READ_MODIFIED | MODIFIED_RESPONSE) & ~INVALID_STATUS & the CACHE_AGENT cannot guarantee INVALID_STATUS at the end of the transaction) and shall maintain ASSERT_CS set until CACHE_OP_CMPLT.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

INTERVENTION. A module shall set INTERVENTION if CACHED & EXCLUSIVE_MODIFIED_STATUS & (READ_SHARED | READ_MODIFIED | READ_INVALID) and shall maintain INTERVENTION set until CACHE_OP_CMPLT.

WAIT_CACHED. A module shall set WAIT_CACHED while CACHED & RESPONDER & (READ_SHARED | READ_MODIFIED | INVALIDATE | READ_INVALID).

SPLIT_CACHED. A module may set SPLIT_CACHED while CACHED & (CACHE_AGENT & (EXCLUSIVE_MODIFIED_STATUS & (READ_SHARED | READ_MODIFIED | READ_INVALID) | ~INVALID_STATUS & (INVALIDATE | WRITE_INVALID)) | MEMORY_AGENT & (READ_SHARED | READ_MODIFIED | INVALIDATE | READ_INVALID | WRITE_INVALID) | MODIFIED_RESPONSE & RESPONDER_STATUS).

QUEUED_TRANSACTION_FLAG. Modules shall set QUEUED_TRANSACTION_FLAG if PARTICIPATING & ~MULTIPLE_PACKET_MODE & DISC_INFO_CAPTURED & **ST4*** | LAST_TRANSACTION_FLAG, and shall maintain it set until CACHE_OP_CMPLT.

INVALID_STATUS. A module shall set INVALID_STATUS if ADDR_DECODED & CACHED and the addressed cache lines state is INVALID. Modules shall maintain INVALID_STATUS set until CACHE_OP_CMPLT.

SHARED_UNMODIFIED_STATUS. A module shall set SHARED_UNMODIFIED_STATUS if ADDR_DECODED & CACHED and the addressed cache lines state is SHARED_UNMODIFIED. Modules shall maintain SHARED_UNMODIFIED_STATUS set until CACHE_OP_CMPLT.

EXCLUSIVE_UNMODIFIED_STATUS. A module shall set EXCLUSIVE_UNMODIFIED_STATUS if ADDR_DECODED & CACHED and the addressed cache lines state is EXCLUSIVE_UNMODIFIED. Modules shall maintain EXCLUSIVE_UNMODIFIED_STATUS set until CACHE_OP_CMPLT.

EXCLUSIVE_MODIFIED_STATUS. A module shall set EXCLUSIVE_MODIFIED_STATUS if ADDR_DECODED & CACHED and the addressed cache lines state is EXCLUSIVE_MODIFIED. Modules shall maintain EXCLUSIVE_MODIFIED_STATUS set until CACHE_OP_CMPLT.

REQUESTER_STATUS. A module shall set REQUESTER_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of REQUESTER. Modules shall maintain REQUESTER_STATUS set until CACHE_OP_CMPLT.

REQUESTER_EXCLUSIVE_STATUS. A module shall set REQUESTER_EXCLUSIVE_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of REQUESTER_EXCLUSIVE. Modules shall maintain REQUESTER_EXCLUSIVE_STATUS set until CACHE_OP_CMPLT.

REQUESTER_SHARED_STATUS. A module shall set REQUESTER_SHARED_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of REQUESTER_SHARED. Modules shall maintain REQUESTER_SHARED_STATUS set until CACHE_OP_CMPLT.

REQUESTER_WRITE_INVALID_STATUS. A module shall set REQUESTER_WRITE_INVALID_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of REQUESTER_WRITE_INVALID. Modules shall maintain REQUESTER_WRITE_INVALID_STATUS until CACHE_OP_CMPLT.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

RESPONDER_STATUS. A module shall set RESPONDER_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of RESPONDER. Modules shall maintain RESPONDER_STATUS until CACHE_OP_CMPLT.

RESPONDER_EXCLUSIVE_STATUS. A module shall set RESPONDER_EXCLUSIVE_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of RESPONDER_EXCLUSIVE. Modules shall maintain RESPONDER_EXCLUSIVE_STATUS until CACHE_OP_CMPLT.

RESPONDER_INVALIDATE_STATUS. A module shall set RESPONDER_INVALIDATE_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of RESPONDER_INVALIDATE. Modules shall maintain RESPONDER_INVALIDATE_STATUS until CACHE_OP_CMPLT.

RESPONDER_SHARED_STATUS. A module shall set RESPONDER_SHARED_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of RESPONDER_SHARED. Modules shall maintain RESPONDER_SHARED_STATUS until CACHE_OP_CMPLT.

RESPONDER_WRITE_INVALID_STATUS. A module shall set RESPONDER_WRITE_INVALID_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of RESPONDER_WRITE_INVALID. Modules shall maintain RESPONDER_WRITE_INVALID_STATUS until CACHE_OP_CMPLT.

REQUESTER_WAITING_STATUS. A module shall set REQUESTER_WAITING_STATUS if ADDR_DECODED & CACHED and the addressed cache line has the attribute of REQUESTER_WAITING. Modules shall maintain REQUESTER_WAITING_STATUS until CACHE_OP_CMPLT.

8.2.3 Cache module attributes per cache line

INVALID. A CACHE or CACHE_AGENT shall set INVALID and clear SHARED_UNMODIFIED | EXCLUSIVE_UNMODIFIED | EXCLUSIVE_MODIFIED for all lines if SYS_RESET.

A CACHE or CACHE_AGENT shall set INVALID and clear SHARED_UNMODIFIED | EXCLUSIVE_UNMODIFIED | EXCLUSIVE_MODIFIED if MASTER & (MODIFIED_RESPONSE | WRITE_INVALID | READ_INVALID | (COPY_BACK | SHARED_RESPONSE) & ~KEEP_COPY) | CACHED & ~KEEP_COPY & (READ_SHARED | READ_INVALID) | CACHED & ~SPLIT & (READ_MODIFIED | INVALIDATE | WRITE_INVALID).

A CACHE or CACHE_AGENT may set INVALID and clear SHARED_UNMODIFIED | EXCLUSIVE_UNMODIFIED if ~EXCLUSIVE_MODIFIED.

A CACHE or CACHE_AGENT shall not allow read or modify access to the data in a cache line if INVALID is set.

SHARED_UNMODIFIED. A CACHE or CACHE_AGENT shall set SHARED_UNMODIFIED and clear INVALID |EXCLUSIVE_UNMODIFIED | EXCLUSIVE_MODIFIED if MASTER & (INVALID_STATUS & ~ADDRESS_ONLY & (READ_SHARED | READ_MODIFIED) | KEEP_COPY & (COPY_BACK | SHARED_RESPONSE)) | CACHED & (REQUESTER_SHARED & SHARED_RESPONSE & INVALID_STATUS & ~ADDRESS_ONLY & QUEUED_TRANSACTION_FLAG | SNARF_DATA & ~ADDRESS_ONLY | REQUESTER_EXCLUSIVE & MODIFIED_RESPONSE & ~ADDRESS_ONLY & SPLIT_STATUS | ~INVALID_STATUS & KEEP_COPY & (ASSERT_CS | QUEUED_TRANSACTION_FLAG)).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A CACHE or CACHE_AGENT may set SHARED_UNMODIFIED and clear EXCLUSIVE_UNMODIFIED if EXCLUSIVE_UNMODIFIED.

A CACHE or CACHE_AGENT shall not allow modify access to the data in a cache line if SHARED_UNMODIFIED is set. A CACHE or CACHE_AGENT may allow read access to the data in a cache line if SHARED_UNMODIFIED is set.

EXCLUSIVE_UNMODIFIED. A CACHE or CACHE_AGENT shall set EXCLUSIVE_UNMODIFIED and clear INVALID | SHARED_UNMODIFIED | EXCLUSIVE_MODIFIED if INVALID_STATUS & ~ADDRESS_ONLY & ~QUEUED_TRANSACTION_FLAG & (MASTER & READ_SHARED | CACHED & REQUESTER_SHARED & SHARED_RESPONSE).

A CACHE or CACHE_AGENT shall not allow modify access to the data in a cache line if EXCLUSIVE_UNMODIFIED is set. A CACHE or CACHE_AGENT may allow read access to the data in a cache line if EXCLUSIVE_UNMODIFIED is set.

EXCLUSIVE_MODIFIED. A CACHE or CACHE_AGENT shall set EXCLUSIVE_MODIFIED and clear INVALID | SHARED_UNMODIFIED | EXCLUSIVE_UNMODIFIED if MASTER & ~SPLIT_STATUS & ~WAIT_STATUS & (INVALID_STATUS & READ_MODIFIED & ~QUEUED_TRANSACTION_FLAG | SHARED_UNMODIFIED_STATUS & INVALIDATE) | CACHED & REQUESTER_EXCLUSIVE & MODIFIED_RESPONSE & ~SPLIT_STATUS & (INVALID_STATUS & ~ADDRESS_ONLY | SHARED_UNMODIFIED_STATUS).

A CACHE or CACHE_AGENT may set EXCLUSIVE_MODIFIED and clear EXCLUSIVE_UNMODIFIED if EXCLUSIVE_UNMODIFIED.

A CACHE or CACHE_AGENT may allow read or modify access to the data in a cache line if EXCLUSIVE_MODIFIED is set.

8.2.4 Requester attributes per cache line

REQUESTER. A module shall set REQUESTER for a cache line if REQUESTER_EXCLUSIVE | REQUESTER_SHARED | REQUESTER_WRITE_INVALID.

REQUESTER_EXCLUSIVE. A module shall set REQUESTER_EXCLUSIVE if MASTER & SPLIT_STATUS & (READ_MODIFIED | INVALIDATE) and shall clear REQUESTER_EXCLUSIVE if CACHED & (~SPLIT_STATUS & MODIFIED_RESPONSE | WRITE_INVALID).

A module shall clear REQUESTER_EXCLUSIVE for all lines if SYS_RESET.

REQUESTER_SHARED. A module shall set REQUESTER_SHARED if MASTER & SPLIT_STATUS & (READ_SHARED | READ_INVALID) and shall clear REQUESTER_SHARED if CACHED & (SHARED_RESPONSE | WRITE_INVALID | COPY_BACK).

A module shall clear REQUESTER_SHARED for all lines if SYS_RESET.

REQUESTER_WRITE_INVALID. A module shall set REQUESTER_WRITE_INVALID if MASTER & SPLIT_STATUS & WRITE_INVALID and shall clear REQUESTER_WRITE_INVALID if CACHED & (~SPLIT_STATUS & MODIFIED_RESPONSE | WRITE_INVALID).

A module shall clear REQUESTER_WRITE_INVALID for all lines if SYS_RESET.

REQUESTER_WAITING. A module shall set REQUESTER_WAITING if MASTER & WAIT_STATUS & (READ_SHARED | READ_MODIFIED | INVALIDATE | READ_INVALID).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A module may set REQUESTER_WAITING if ~MASTER & (SPLIT_STATUS | WAIT_STATUS) & (READ_SHARED | READ_MODIFIED | INVALIDATE | READ_INVALID | MODIFIED_RESPONSE).

A module shall clear REQUESTER_WAITING if CACHED & (SHARED_RESPONSE | MODIFIED_RESPONSE & ~SPLIT_STATUS | WRITE_INVALID | COPYBACK).

A module shall clear REQUESTER_WAITING for all lines if SYS_RESET. Modules may clear REQUESTER_WAITING if a user defined timeout occurs.

8.2.5 Responder attributes per cache line

RESPONDER. A module shall set RESPONDER if RESPONDER_EXCLUSIVE | RESPONDER_INVALIDATE | RESPONDER_SHARED | RESPONDER_WRITE_INVALID.

RESPONDER_EXCLUSIVE. A module shall set RESPONDER_EXCLUSIVE if CACHED & (SPLIT & EXCLUSIVE_MODIFIED_STATUS & READ_MODIFIED | SPLIT & MEMORY_AGENT & **~ST5*** & READ_MODIFIED) | MEMORY_AGENT & RESPONDER_INVALIDATE_STATUS & INVALIDATE_COLLISION.

A module shall clear RESPONDER_EXCLUSIVE if MASTER & MODIFIED_RESPONSE | CACHED & WRITE_INVALID.

A module shall clear RESPONDER_EXCLUSIVE for all lines if SYS_RESET.

RESPONDER_INVALIDATE. A module shall set RESPONDER_INVALIDATE if CACHED & (CACHE_AGENT | MEMORY_AGENT) & (INVALIDATE | READ_MODIFIED & SPLIT_STATUS) & ~INVALID_STATUS & the CACHE_AGENT can not guarantee INVALID_STATUS at the end of the transaction.

A module shall clear RESPONDER_INVALIDATE if (MASTER | CACHED & ~SPLIT) & MODIFIED_RESPONSE & (CACHE_AGENT | MEMORY_AGENT) & INVALID_STATUS | CACHED & WRITE_INVALID | MEMORY_AGENT & INVALIDATE_COLLISION.

A module shall clear RESPONDER_INVALIDATE for all lines if SYS_RESET.

RESPONDER_SHARED. A module shall set RESPONDER_SHARED if CACHED & SPLIT & (READ_SHARED | READ_INVALID).

A module shall clear RESPONDER_SHARED if MASTER & SHARED_RESPONSE | WRITE_INVALID.

A module shall clear RESPONDER_SHARED for all lines if SYS_RESET.

RESPONDER_WRITE_INVALID. A module shall set RESPONDER_WRITE_INVALID if CACHED & SPLIT & WRITE_INVALID.

A module shall clear RESPONDER_WRITE_INVALID if (MASTER | CACHED & ~SPLIT) & MODIFIED_RESPONSE | WRITE_INVALID.

A module shall clear RESPONDER_WRITE_INVALID for all lines if SYS_RESET.

INVALIDATE_COLLISION. Invalidate collisions occur in bus bridges and are therefore not within the scope of this specification. INVALIDATE_COLLISION shall be set when an invalidate from one bus

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

collides with an invalidate from another bus. INVALIDATE_COLLISION shall be cleared when the invalidate collision is resolved.

8.2.6 Protocol definition

8.2.6.1 Cache line size

Cache lines shall be 64 contiguous bytes of memory aligned modulus 64.

8.2.6.2 Cache line transfers

All cache line transfers shall begin with the first word of the cache line and complete when the entire cache line has been transferred. Partial cache line transfers shall not be performed and the complete cache line shall be transferred within a single transaction.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9. Message passing

9.1 Description

Futurebus+ message passing provides simple and efficient communications between processors in Futurebus+ systems. At the application (or system) level, processes in one module of a Futurebus+ system send messages to or receive messages from processes in other modules. The message passing levels necessary to support a large variety of operating environments are the subject of the message-passing specification.

Futurebus+ message passing is specified at two levels:

1) The *frame level* provides the basic protocol for the transfer of a logical entity called a message frame between modules within a Futurebus+ system.
2) The *message level*, using the message frames defined in the frame level, transports variable length messages between modules.

Modules with message-passing capability use several predefined addresses within the control and status register space as mailboxes to which message frames are transferred. Because only Futurebus+ message frames are written to CSR message mailboxes, they are easily differentiated from and compatible with all other Futurebus+ transactions as shown in figure 53.

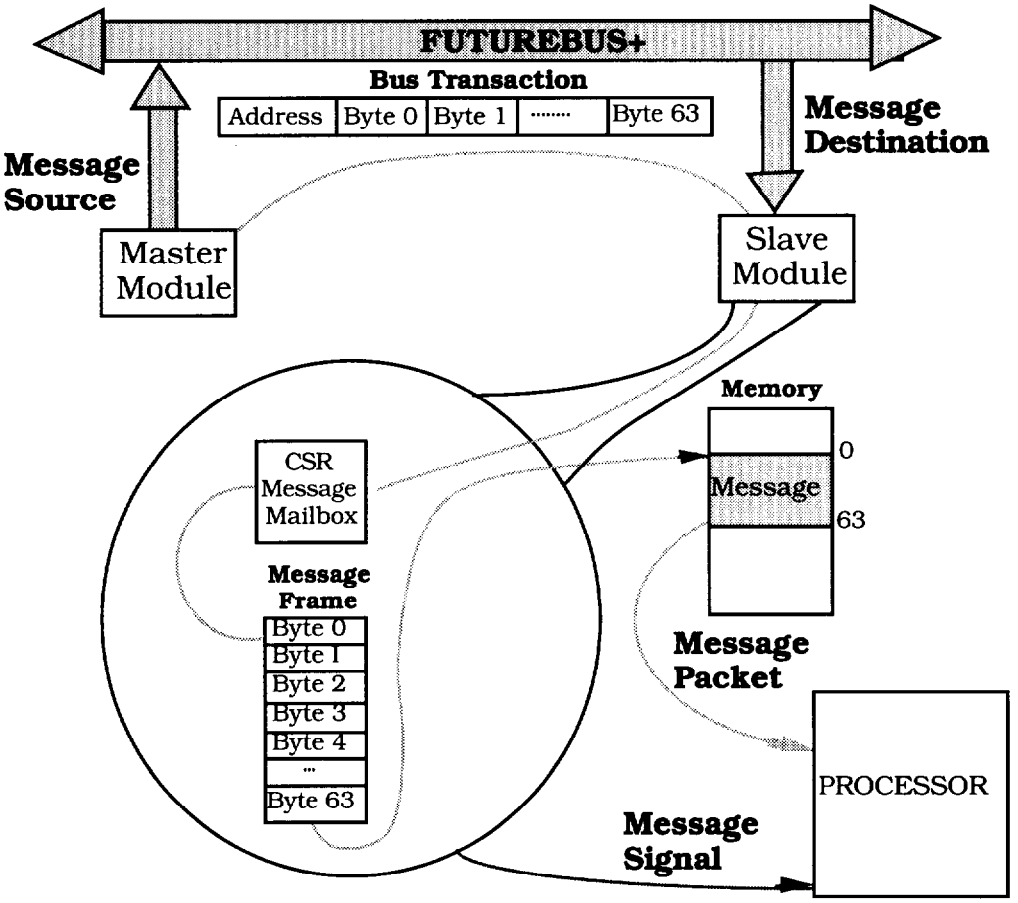


Figure 53—Futurebus+ message passing

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.1.1 Frame level

A message frame is a block of data sent by a master to a slave mailbox using a write transaction. Write unlocked or write no acknowledge transactions may be used. The write no acknowledge transaction command is an optional feature of Futurebus+ message passing. The mailbox can be thought of as a "magic" location. It does not represent a memory location, but a window through which messages pass. The above addressing scheme allows message passing modules to communicate with each other independent of processor type or operating environment. Once a message frame has been received, a module processes the frame based on the information contained in its header.

9.1.1.1 Frame structure

Because message frames can be transferred between different processor types with different internal representations of data, the message frame will be defined as it appears on the bus. As shown in figure 54, the various quadlets and bytes making up a message frame are defined for how they would be seen on 32, 64, 128, and 256-bit buses.

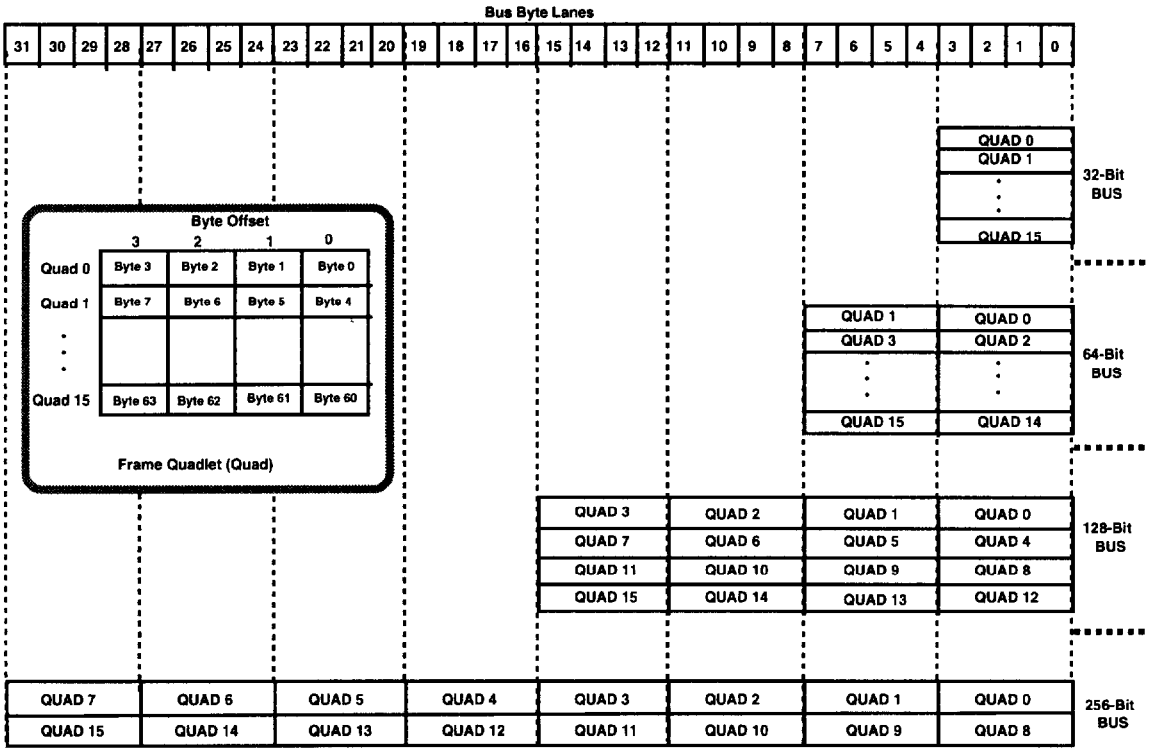


Figure 54—Frame notation

As shown in figure 55, a 64-byte message frame is composed of a header and a body and is transferred over Futurebus+ to a message mailbox. The first four bytes of a frame contains header information which is common to all Futurebus+ message frames and defines the higher level protocol to be used. The body of the frame consists of 60-bytes of frame data.

There are four fields in the frame header:

1) The write command type selects the write transaction command to be used. If the field is a zero, a fully connected (write unlocked transaction) command is used. If the field is a one, a unconnected (write no acknowledge) command is used.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

a) The order of write unlocked transactions is preserved within a backplane. The order of write unlocked transactions may not be preserved over a bus bridge if the bridge is splitting transactions.
b) The order of write no acknowledge transactions is not guaranteed. It is the responsibility of the message protocols to guarantee message integrity.

2) The message protocol field selects the higher level protocol used to process the frame. The message protocol field is defined as follows:
   a) If the value of the message protocol field is zero, the message frame is not part of a higher level protocol.
   b) If the value of the message protocol field is one, the message frame is part of a message that complies with the Futurebus+ message level protocol.
   c) If the value of the message protocol field is 2–63, the message level protocol is not defined by this International Standard but reserved for future use.
   d) If the value of the message protocol field is 64–79, the message complies with a message level protocol defined in a related standard.
   e) If the value of the message protocol field is 80–127, the message complies with a vendor-dependent message level protocol.
3) The message priority field defines the priority associated with the message frame. The message priority field is the same as the priority found in the disconnection phase of the message frame's write transaction. The actual values used in the priority field are beyond the scope of this specification.
4) The global identification field (global identification lower and global indentification upper) identifies the module sending the message frame. As it appears on the bus, the global identification field is the same as the global identification as described in 6.1.17.11.

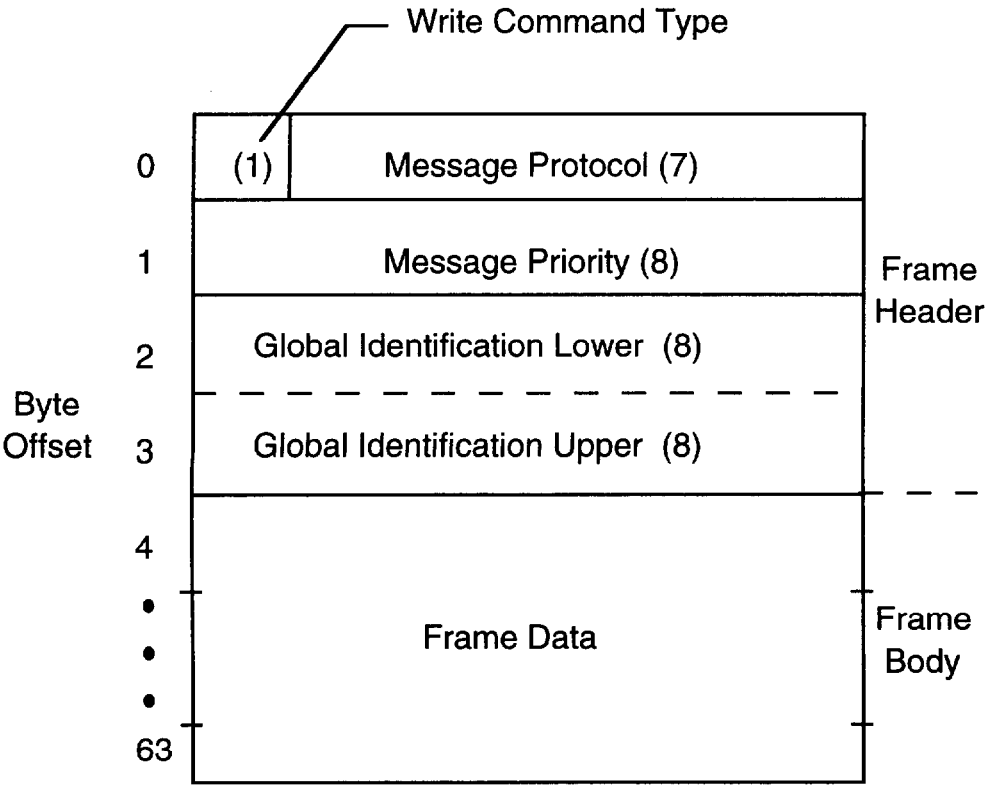


Figure 55—64-Byte frame format

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Because each message frame has a message protocol value, several message-passing protocols may be used within the same system without interfering with each other. If a module receives a message frame with a message protocol that is not supported by that module, the module may disregard the message frame.

The default frame size (MESSAGE_FRAME_SIZE) is 64-bytes. All Futurebus+ message-passing modules will be able to transmit and receive 64-byte messages using a 32-bit bus width and compelled write unlocked transactions. Optional frame sizes, bus widths, and modes of transfers are supported if both sending and receiving modules support the features. Modules capable of larger frames or bus widths are downward compatible with modules having a default frame size and bus width. Message frames with sizes smaller than 64-bytes can be transferred between modules when the message protocol supports this feature. A module must avoid sending larger frames to modules than they support, since a transaction error will result. Message frames may be transferred in packet mode if the frame size is consistent with an allowed packet size and the slave module is packet capable.

The default maximum frame size of 64 bytes was selected to maintain compatibility with other aspects of Futurebus+ (cache line size and packet mode transfers). Larger frames may optionally be used as the need arises. Since a buffer for a large frame may be used to receive a small frame but not vice versa, interoperability between modules having two different frame sizes must be assured. As described in 7.2, each module that supports message passing has a message frame size capability register so it can be determined which frame sizes will be compatible.

9.1.1.2 Mailbox structure

Each module which supports message passing has a mailbox structure similar to that shown in figure 56. In order to avoid deadlock situations, caused by the use of a single mailbox, each message-passing module has a request and a response mailbox. Mailboxes can receive broadcast as well as single-slave transactions.

The message mailboxes offer a mechanism for differentiating a message transaction from other Futurebus+ transactions. The addresses for the message mailboxes are beyond the scope of this International Standard.

Message-passing slave modules have facilities to receive a frame from the bus and pass it on to a storage device such as a hardware queue, memory, cache line, or register.

An implementation is allowed to merge the request and response mailboxes if deadlock-free operations can be maintained. Modules that merge the request and response mailboxes must recognize both addresses.

Implementations requiring the preservation of ordering for message transactions can do so by using a single mailbox (request or response). Message level protocols that preserve the order of message transactions by using a single message mailbox must also make provisions for deadlock-free operations.

9.1.1.2.1 Request mailbox

The *request mailbox* and its associated queue are used for message frames that require an acknowledgement message to be returned to the requesting module. If the master sends a message frame to the request mailbox of a slave module, the slave module will be expected to return a response message to the master's response mailbox as indicated by the protocol.

9.1.1.2.2 Response mailbox

Acknowledgement frames are sent to a requesting module's *response mailbox.*

The response mailbox and its associated queue are used for message frames that do not require an acknowledgement. If a master module sends a message frame to the response mailbox of a slave module, a return

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

message is not required. Messages sent in this manner are point-to-point in nature and have only one destination.

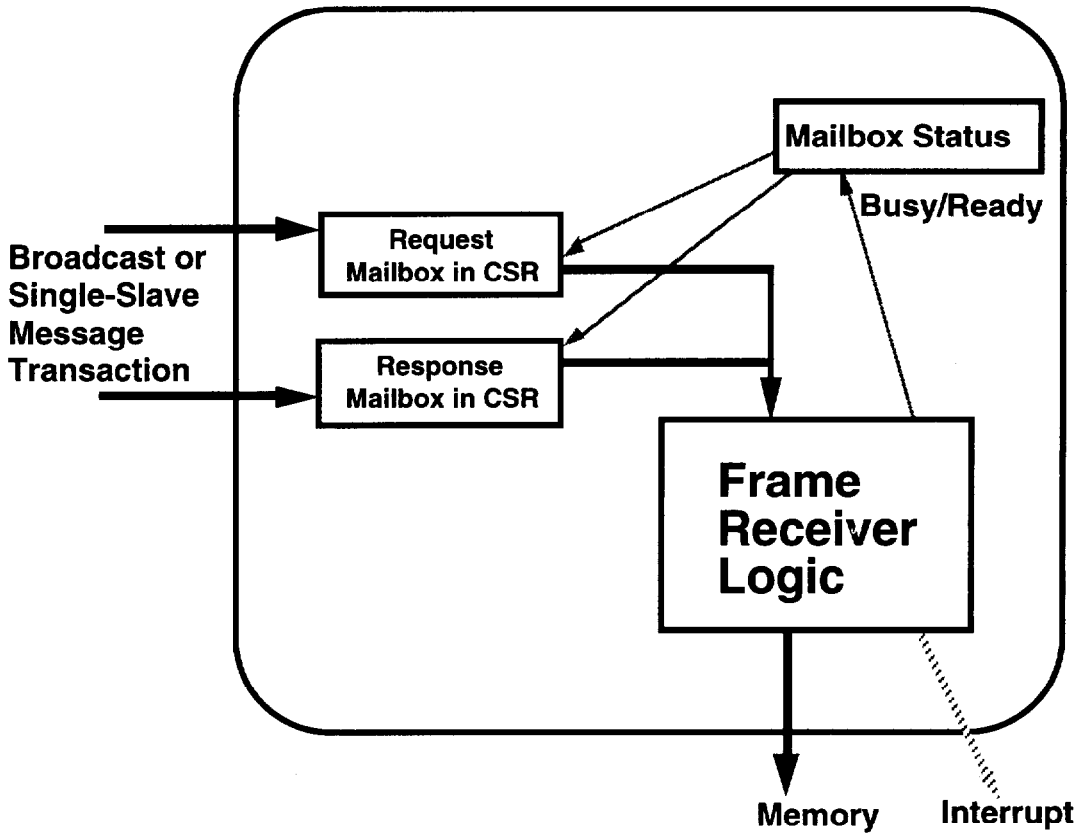


Figure 56—Message passing mailboxes

9.1.1.2.3 Broadcast transactions

Broadcast transactions can be sent to the response mailboxes of all message-passing modules on the bus. Broadcast messages should not be acknowledged. Modules receiving a broadcast message are required to induce broadcast mode by asserting **bc***. Broadcast transactions are sent to an unique module address.

Message frames that are broadcast have attributes that allow them to be selectively received, as shown in figure 57. When decoding the address of a broadcast transaction, modules utilize the least significant six bits of the address. If the value of the least significant six bits of the address is zero, the slave always receives the message. If the value of the least significant six bits of the address is not zero, they are used to address one of 63 broadcast mailbox select attributes contained in the select mask register as described in 7.1 (MESSAGE_PASSING_SELECT_MASK). If an addressed select attribute is clear, a module ignores the transaction and does not assert **sl*** in the connection phase. By setting the appropriate select attribute, message frames are effectively filtered before they can enter the module. This feature of Futurebus+ message passing allows messages to be broadcast over Futurebus+ and yet allows slaves to selectively receive those messages.

Any slave whose mailbox is busy will cause the transaction to be terminated with a **bs*** status. Broadcast frames are very efficient for those applications that require many modules to receive the same messages at the same time. However, indiscriminate use of broadcast frames may cause performance problems and

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

excessive retries due to busy mailboxes. By use of select attributes, the probability of encountering a busy mailbox may be reduced.

It is recommended that message broadcasts be used in moderation unless special provisions are made to minimize the probability that the mailbox is busy.

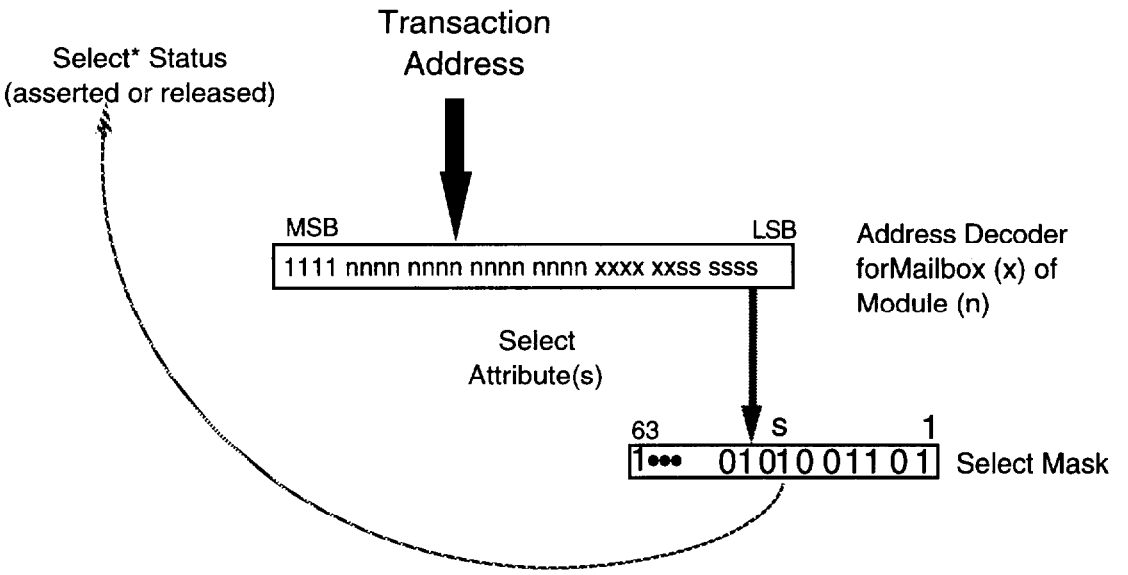


Figure 57—Broadcast mailbox select attribute

9.1.1.3 Frame level protocol

Write unlocked or write no acknowledge transactions can be used to transfer a message frame to a message mailbox. An address only unlocked transaction may be used to determine the current status of a mailbox. Address only transactions will not change the state of a mailbox (e.g., busy or message conflict). Modules assert **be*** to indicate an error if any other transaction type is received in a transaction to a mailbox.

Modules that support message passing assert **sl*** in response to all transactions to request and response mailboxes. Message-passing modules assert **sl*** and **bc*** in response to all transactions to the broadcast mailboxes if the appropriate select attribute is set or **AD[5..0]*** are released.

If the master attempts to send a frame that is larger than the slave module supports, the slave will assert **te***, and the master on seeing **TE*** asserted should interpret this as an error. Slaves assert **ed*** during the last transfer that it can accommodate. If a master receives **ED*** status it must end the transaction because the slave module can accept no more data.

If a module receives a transaction to a mailbox but is temporarily unable to receive the data it asserts **bs***. The originating module should initiate a retry at a later time. Modules use busy status to prevent unprocessed frames from being overwritten by incoming frames.

A message-passing module will assert or release **tf*** by setting or clearing MESSAGE_CONFLICT.

After a system reset, a message-passing module will set MESSAGE_CONFLICT for all transactions to any of its message mailboxes until the module's message mailbox is properly initialized.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

After the module's message mailbox is properly initialized, it will set MESSAGE_CONFLICT when it is not able to receive a message frame due to a resource limitation that will not likely become available in a timely fashion (much greater than busy retry delay).

Upon receiving a **TF*** status for a message-passing transaction, the sending module will abort the transaction. It is the responsibility of a higher level recovery protocol in the sending module to attempt a retry at a time when it is more likely that the message transaction will be successful (much greater than busy retry delay).

9.1.1.4 Message priority

Each message frame has a priority field. How the frame priority is used to arbitrate for the bus is beyond the scope of this specification.

Where queues are used, message frames in both transmission and receive queues should be ordered in priority order. Frames with the same priority should be ordered on a first-in, first-out basis. If a short hardware transmission queue is used and the queue is fully occupied by lower priority frames (leading to priority inversion), mechanisms must be provided to allow for the pre-emption of a lower priority frame in favor of a high priority frame.

9.1.1.5 Frame level example

An example of a possible implementation of frame receiver logic is shown in figure 58. It is based on the linkage of message blocks using a linked list mechanism providing an efficient method of receiving sequences of frames. The process of receiving frames is as follows:

1) Prior to receiving any messages, the frame pointer register and the linked list pointer point to the same memory location.
2) Another module then transmits a message frame to the module's frame input buffer.
3) The message-passing control logic transfers the data from the frame input buffer to the block of memory pointed to by the linked list pointer.
4) Using the next frame pointer, the frame pointer register is then changed to point at the next entry in the linked list.
5) The message passing control logic then notifies the processor that a message frame has been received and resets **bs***.
6) A sending module then transmits another message frame to the modules frame input buffer.
7) While the message passing control logic transfers the data from the frame input buffer to the block of memory pointed to by the linked list pointer, a module sends another message to the module.
8) The receiving module asserts **bs***, indicating that the module should retry later.
9) When the transfer from the frame input buffer is complete, the frame pointer register is then changed to point at the next entry in the linked list.
10) The message-passing control logic then notifies (e.g., interrupts) the processor that another message frame has been received.
11) The sending module tries again and transmits a message frame to the modules frame input buffer.
12) The message-passing control logic transfers the data from the frame input buffer to the block of memory pointed to by the linked list pointer.
13) The next frame pointer is zero at that point, so the frame pointer register is set to zero.
14) The message-passing control logic then notifies the processor that a message frame has been received.
15) At any step after step 5), the processor can remove messages from the linked list starting with the message pointed to by the linked list pointer. After a message is removed, the linked list pointer is updated with the next frame pointer. This process can be continued until the linked list pointer is equal to the frame pointer register. In addition, the processor can add more empty frames onto the end of the linked list.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

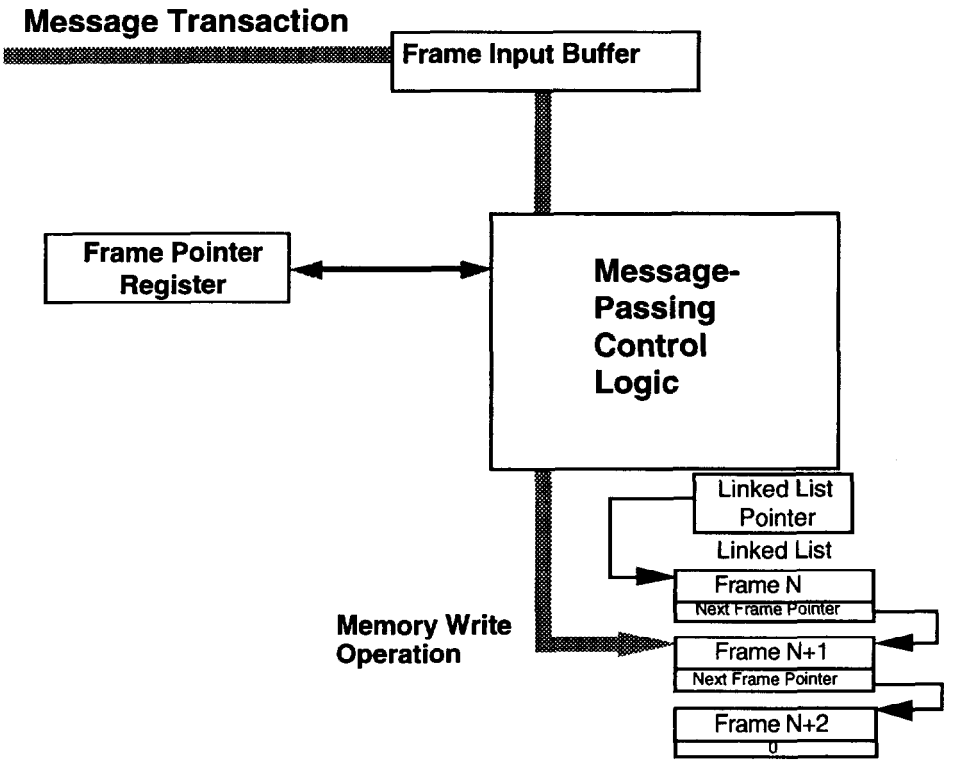


Figure 58—Linked list message passer

9.1.2 Message level

The following clauses apply when the message protocol field defined in the frame header is equal to one.

9.1.2.1 Message level frame structure

A message is a logical entity that consists of one or more message frames. A message level frame consists of an eight or twelve byte header followed by a variable number of bytes of frame data, as shown in figure 59 (For illustrative purposes, the message level frame is shown as a 64-byte frame). The contents of the message level frame are determined by the frame type.

The message protocol field of the message level frame is the same as in the frame level. The message protocol field is a value of one for Futurebus+ message level messages. The message priority and global identification fields of the frame are the same as in the frame level.

The following subclauses explain the remainder of the message level header.

9.1.2.1.1 Frame type

The frame type field indicates that the frame is one of the following types:

1) Unacknowledged event frame
2) Acknowledged event frame
3) Event positive acknowledge frame
4) Event negative acknowledge frame
5) Multiple frame message request frame
6) Multiple frame message positive response frame

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

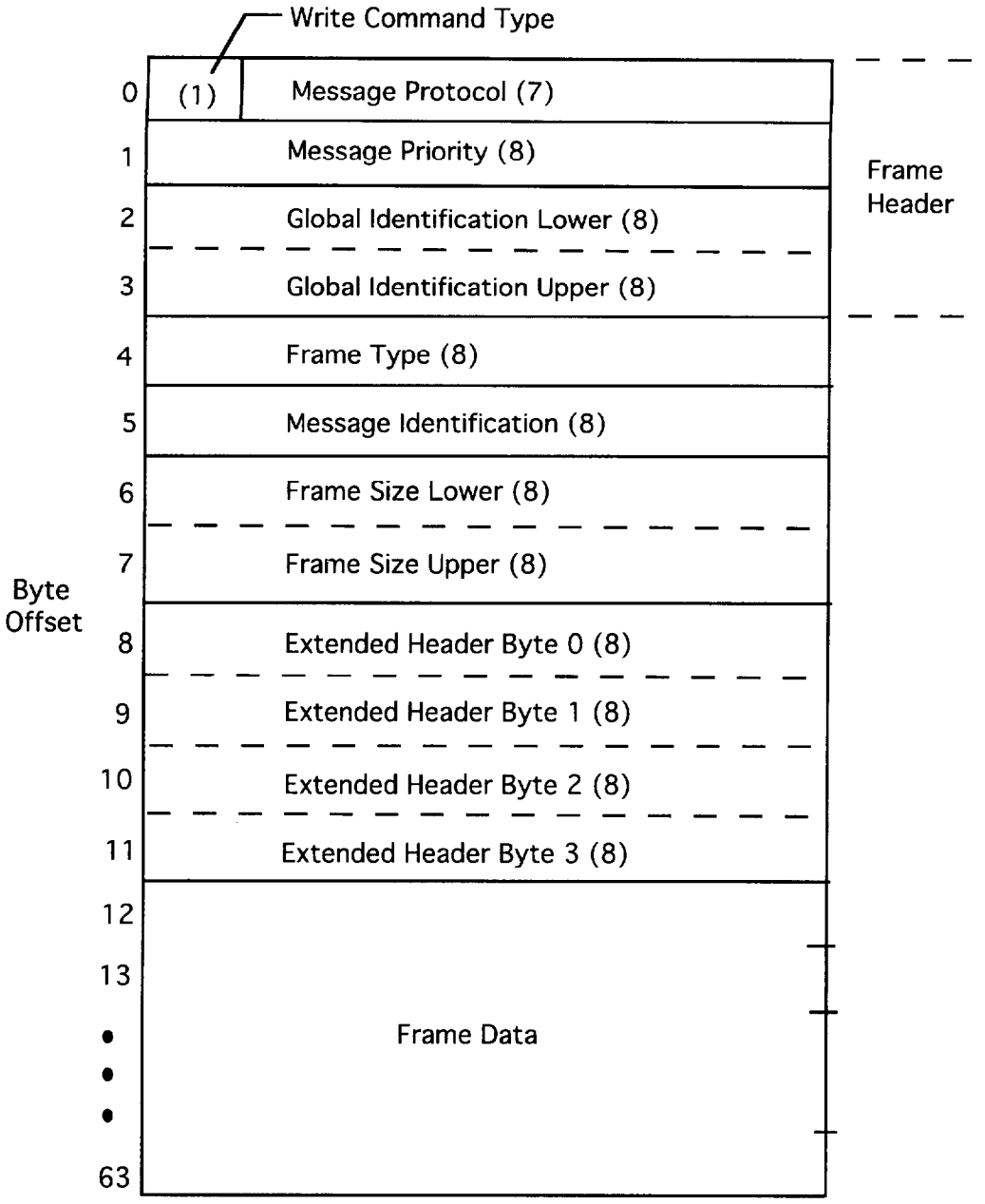


Figure 59—Futurebus+ message level frame structure

7) Multiple frame message negative response frame
8) Multiple frame message ordered data frame
9) Multiple frame message sequenced data frame
10) Multiple frame message last ordered frame
11) Multiple frame message last sequenced frame
12) Multiple frame message positive acknowledge frame
13) Multiple frame message negative acknowledge frame

Other frame types are reserved for future use.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If the most significant bit of the frame type is set, an attention request event (i.e., interrupt) will be generated in the module receiving the frame.

All of the frame types are compatible with the write unlocked transaction. All of the frame types except the multiple frame message ordered data frame, are compatible with the write no acknowledge transaction. Sequenced data frames are required only if the optional write no acknowledge command is used. Only unacknowledged event frames can be sent to the broadcast mailbox.

9.1.2.1.2 Frame size

The frame size fields (frame size lower and frame size upper) indicate the total number of bytes used by the frame (frame header, message header, and message body). Although the maximum frame size is determined by a module's hardware, shorter frames may also be transferred. The minimum frame size is eight bytes. This allows modules to send a full range of frame sizes to other modules.

9.1.2.1.3 Message identification

The message-identification field provides a unique identifier for tracking messages. With it a module can initiate several single and multiple frame messages to one or more destination modules. A destination module uses the message identification in conjunction with the global identification to manage its receive buffers. Since each frame in a multiple frame message will have the same message identification value, a mixture of frames from more than one source can be correlated with their appropriate message buffers.

9.1.2.1.4 Extended header

The extended header in the message level header is optional. The contents of the extended header varies depending on the frame type. The extended header consists of Extended Header Byte 0 through Extended Header Byte 3 in order of significance where Extended Header Byte 0 is the least significant byte.

When the frame type does not require an extended header, the extended header fields become part of the frame data field.

9.1.2.1.5 Message size

In some frame types, the extended header is the message size. The message size is the sum of all the bytes in the message passed to the message level from the application level. In an ordered data multiple frame message, it is the sum of all the bytes in the bodies of the ordered data frames and the last ordered frame. In a sequenced multiple frame message, it is the sum of all the bytes in the bodies of the sequenced data frames and the last sequenced frame.

9.1.2.1.6 Frame interval

In some frame types, the extended header is the frame interval. The frame interval is sent to the requesting module to indicate the minimum time between frames of a multiple frame message. Intervals shorter than this may result in the receiving mailbox being busy. The minimum time between frames will be determined largely by whether there is hardware assist for multiple frame messages.

9.1.2.1.7 Sequence number

In some frame types, the extended header is the sequence number. The sequence number is a monotonically increasing number. The first frame in a series is assigned a sequence number of one (1).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.1.2.1.8 Exception type

In some frame types, the extended header is the exception type. The following exception types are defined for all response frames:

1) Invalid frame type—The received frame type was either undefined or received out of order.
2) Invalid frame size—Reception of a frame size larger than that specified in the MESSAGE_FRAME_SIZE CSR was attempted.
3) Invalid message size—A multiple frame message was received but the number of bytes received does not agree with the message size in the request frame.
4) Invalid message identifier—A Futurebus+ message level frame was received by a requester or responder with an unknown message identification.
5) No sequenced frames—A sequenced multiple frame message was received by a module with no support for sequenced frames.
6) Timeout—All of the frames of a multiple frame message were not received within the anticipated time interval.

9.1.2.1.9 Unacknowledged event frame

The unacknowledged event frame is used to send fixed amounts of data to a response mailbox. The body of the frame is used to transfer data. The receiver of this frame will not acknowledge its reception with a response frame. A message identification is unnecessary for this frame type.

9.1.2.1.10 Acknowledged event frame

The acknowledged event frame is used to send fixed amounts of data to a request mailbox. The body of the frame is used to transfer data. The receiver of this frame will acknowledge the reception of this frame by sending an event positive acknowledge frame if the acknowledged event frame is received successfully. The receiver of this frame will acknowledge the reception of this frame by sending an event negative acknowledge frame if the acknowledged event frame is not received successfully.

9.1.2.1.11 Event positive acknowledge frame

The event positive acknowledge frame is sent to a response mailbox of the module originating an acknowledged event frame was successfully received and processed. The message identification of the event positive acknowledge frame must be the same as the message identification received in the acknowledged event frame. The remainder of the frame is unused.

9.1.2.1.12 Event negative acknowledge frame

The event negative acknowledge frame is sent to a response mailbox of the module originating an acknowledged event frame that was not successfully processed. The message identification of the event positive acknowledge frame must be the same as the message identification received in the acknowledged event frame. The extended header is used with this message type to indicate the reason for the failure. The remainder of the frame can be used for additional information about the failure.

9.1.2.1.13 Multiple frame message request frame

To transfer more data than can be contained in a single frame, a module may initiate a multiple frame message sequence. The multiple frame message request frame is sent to a request mailbox to initiate a multiple frame message. The extended header contains the message size of the multiple frame message. The message identification is used as a unique identifier of the multiple frame message sequence. The module originating a multiple frame message frame request will receive either a multiple frame message positive response

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

frame or a multiple frame message negative response frame in response. The remainder of the frame is unused.

9.1.2.1.14 Multiple frame message positive response frame

The multiple frame message positive response frame is sent to a response mailbox if conditions allow it to honor a multiple frame message request frame. The message identification of the response frame must be the same as the message identification received in the request frame. The extended header is used with this message type to indicate the frame interval. The remainder of the frame is unused.

9.1.2.1.15 Multiple frame message negative response frame

The multiple frame message negative response frame is sent to a response mailbox if conditions do not allow it to honor a multiple frame message request frame. The message identification of the response frame must be the same as the message identification received in the request frame. The extended header is used with this message type to indicate the exception code. The remainder of the frame can be used for additional information about the failure.

9.1.2.1.16 Multiple frame message ordered data frame

When the module initiating a multiple frame message receives a positive response frame, it should begin sending ordered data frames to the request mailbox of the receiving module if write unlocked transaction commands are selected. In order to prevent the saturation of the receiving module, an inter-frame interval specified by the frame interval will pass before a subsequent ordered data frame is sent. Each ordered data frame will contain a message fragment. The receiver of these message fragments will assemble the message. The message identification will be the same as in the multiple frame message request frame that initiated the message sequence. The message identification will be used to associate a message fragment with its buffer. The order of the ordered data frames will be guaranteed throughout the system. This frame type is only compatible with write unlocked transactions. The write unlocked transaction is optimum for messages local to the bus.

9.1.2.1.17 Multiple frame message sequenced data frame

When the module initiating a multiple frame message receives a positive response frame, it should begin sending sequenced data frames to the request mailbox of the receiving module if write no acknowledge transaction commands are selected. In order to prevent the saturation of the receiving module, an inter-frame interval specified by the frame interval must pass before a subsequent sequenced data frame is sent. Each sequenced data frame will contain a message fragment. The receiver of these message fragments will assemble the message. The message identification will be the same as in the multiple frame message request frame that initiated the message sequence. The message identification will be used to associate a message fragment with its buffer. A monotonically increasing sequence number will be placed in the extended header for each sequenced data frame. The first sequenced data frame will have a sequence number of one (1). There is no requirement to guarantee the order of the sequenced data frames.

NOTE—Although this frame type was designed specifically for use with the write no acknowledge command, it is compatible with the write unlocked transaction command. In addition, since the write no acknowledge command is optional, sequenced frames are also optional. The write no acknowledge transaction is optimal for bridged systems.

9.1.2.1.18 Multiple frame message last ordered frame

A multiple frame message last ordered frame is sent to the request mailbox by the requesting module to indicate the last frame of an ordered multiple frame message. When a module receives all of the multiple frame message data frames, it will verify that the requested number of bytes have been received. The message identification will be the same as in the multiple frame message request frame that initiated the message

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

sequence. The receiving module will send an acknowledge frame to the requesting module's response mailbox indicating the success or failure of the multiple frame message.

9.1.2.1.19 Multiple frame message last sequenced frame

A multiple frame message last sequenced frame is sent to the request mailbox by the requesting module to indicate the last frame of a sequenced multiple frame message. When a module receives this frame, it will indicate the final sequence number of the sequenced multiple frame message in the extended header. The message identification will be the same as in the multiple frame message request frame that initiated the message sequence. The receiving module will send an acknowledge frame to the requesting module's response mailbox indicating the success or failure of the multiple frame message.

9.1.2.1.20 Multiple frame message positive acknowledge frame

The multiple frame message positive acknowledge frame is sent to a response mailbox if the multiple frame message has been successfully received. The message identification of the acknowledge frame must be the same as the message identification received in the request frame. The remainder of the frame is unused.

9.1.2.1.21 Multiple frame message negative acknowledge frame

The multiple frame message negative acknowledge frame is sent to a response mailbox if the multiple frame message has not been successfully received. The message identification of the acknowledge frame must be the same as the message identification received in the request frame. The extended header is used with this message type to indicate the exception code. The remainder of the frame can be used for additional information about the failure.

A negative acknowledge frame can be sent at any time during a multiple frame message sequence and will cause the sequence to be terminated.

9.1.2.2 Message protocols

9.1.2.2.1 Unacknowledged single frame protocol

An unacknowledged single frame message simply consists of a single message frame being transmitted to a response mailbox as shown in figure 60.

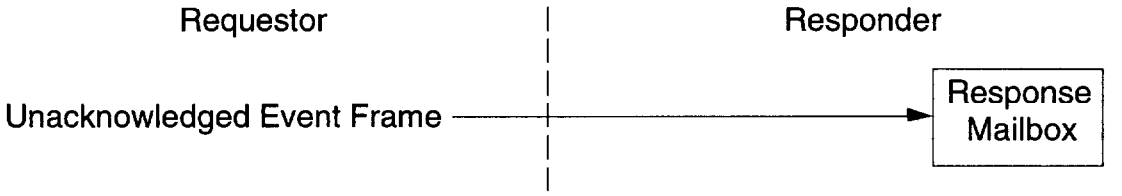


Figure 60—Unacknowledged single frame message

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.1.2.2.2 Acknowledged single frame protocol

An acknowledged single frame message consists of two steps as shown in figure 61:

1) First, an acknowledged event frame is transmitted to the request mailbox of a module.
2) Second, the module receiving the acknowledged event frame responds with an acknowledge frame indicating that the message was received successfully.

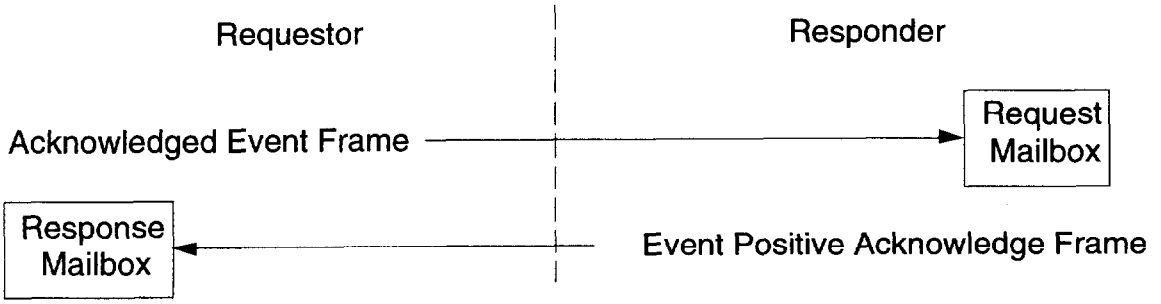


Figure 61—Acknowledged single frame message

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.1.2.2.3 Ordered multiple frame protocol

As shown in figure 62, an ordered multiple frame message proceeds as follows:

1) A multiple frame message request frame is sent to the responder's request mailbox. The request frame will indicate that an ordered multiple frame message is requested. The request frame will also indicate the size of the message.
2) The responder then sends a positive response frame to the requestor's mailbox to indicate it can accept the message. It will also indicate the maximum rate at which the requestor should send frames.
3) The requestor then sends ordered data frames to the responder.
4) The requestor sends zero or more full frames until the remaining data fits into one frame. It then sends a last ordered frame. This notifies the responder that the message transmission has been completed.
5) The responder then sends a positive acknowledge frame to the requestor to indicate that it has received the message correctly.

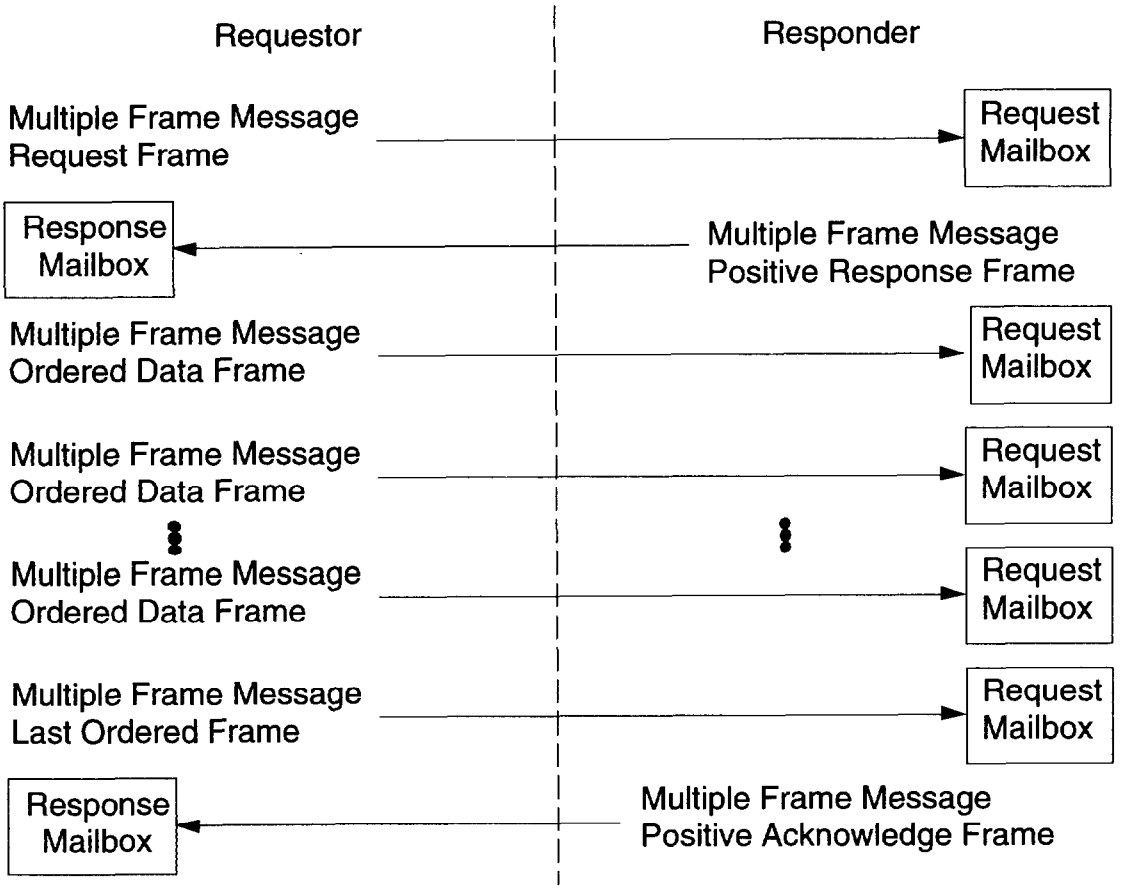


Figure 62—Ordered multiple frame message

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

9.1.2.2.4 Sequenced multiple frame protocol

As shown in figure 63, a sequenced multiple frame message proceeds as follows:

1) A multiple frame message request frame is sent to the responder's request mailbox. The request frame will indicate that a sequenced multiple frame message is requested. The request frame will also indicate the size of the message.
2) The responder then sends a positive response frame to the requestor's mailbox to indicate it can accept the message. It will also indicate the maximum rate at which the requestor should send frames.
3) The requestor then sends sequenced data frames to the responder. The requester will include a sequence number in the extended header of each data frame. This will allow the responder to order the frames after they are received.
4) The requestor sends zero or more full frames until the remaining data fits into one frame. It then sends a last sequenced frame. This notifies the responder that the message transmission has been completed.
5) The responder then sends a positive acknowledge frame to the requestor to indicate that it has received the message correctly.

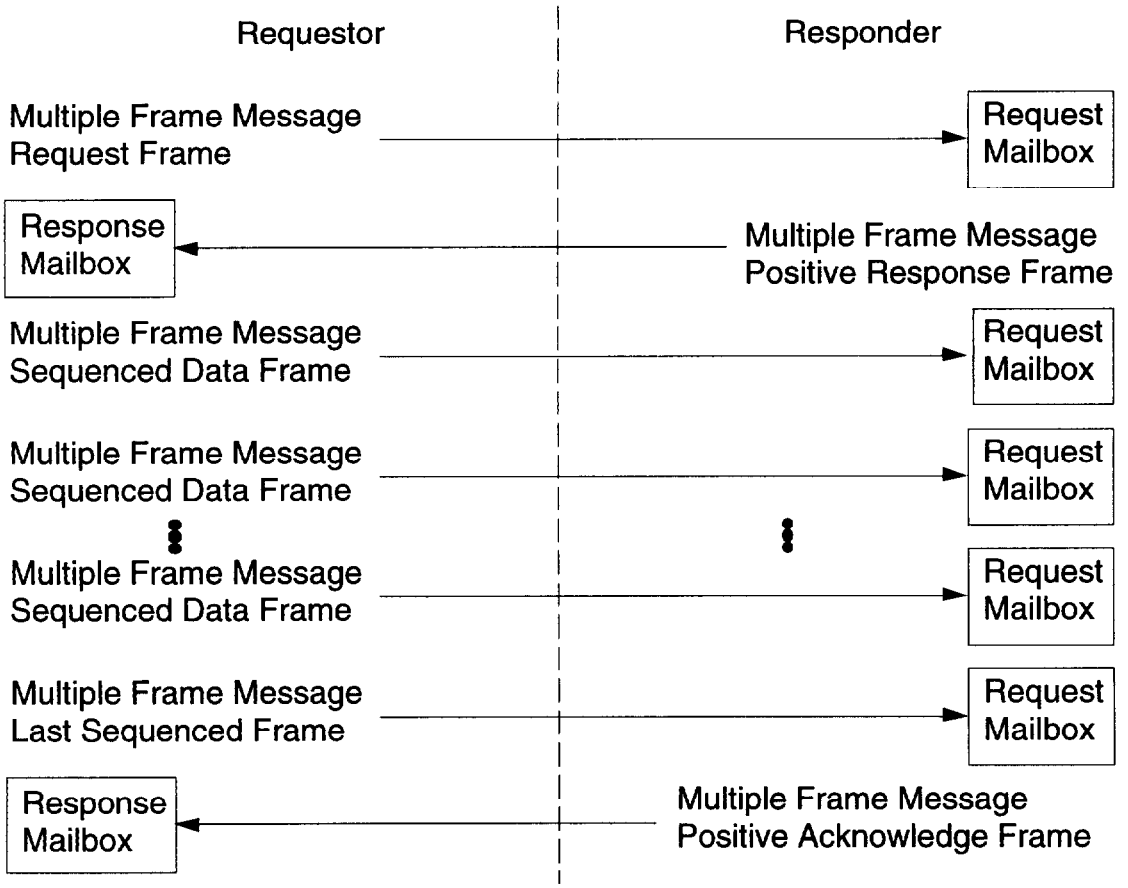


Figure 63—Sequenced multiple frame message

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.1.2.3 Request/response timeout

On a frame-by-frame basis, a sending module should establish a maximum time that can expire between a request and a response frame. If that timeout is exceeded, an error will be passed to the application level. The value of the maximum time is beyond the scope of this International Standard.

9.1.2.4 Multiple frame message timeout

The requesting and responding modules participating in a multiple frame transfer will establish a maximum time that can elapse from the start of a multiple frame message to the successful or unsuccessful completion. If the timeout is exceeded, the multiple frame sequence will be cancelled with a frame indicating a timeout exception. All frames associated with a multiple frame message that times out must be purged. The module requesting the multiple frame message may retry the sequence. The same message identification is used for the retry as was used for the original multiple frame message attempt. The value of the maximum time is beyond the scope of this International Standard.

9.1.2.5 Frame ordering

The message level protocol assumes that frame ordering is preserved when there is a relationship between message frames. Frame ordering is not necessary when there is no relationship between message frames. When there is a relationship, such as the case with solicited messages, frame ordering must be preserved. The preferred mechanism for preserving ordering is to assign the related frames the same priority. Write unlocked transactions are generally used when ordering is important.

9.1.2.6 Bus tenure time

All message-passing module users should be cautioned from initiating message transactions with long tenures, since other transactions with higher priorities may be prevented from being transmitted. By defining a higher level protocol which breaks large messages into a series of shorter message frames, a sequence of frames can be transferred between modules. At the conclusion of each frame transaction, the bus may be relinquished to a higher priority transaction. If another master has not arbitrated for the bus, the next frame can be sent. In this way, large messages can be transferred efficiently without the penalties associated with long bus tenures.

9.2 Specification

9.2.1 Attribute definition

MAILBOX_INITIALIZATION. A module shall set MAILBOX_INITIALIZATION if INIT and shall maintain it set until ~REQUEST_MAILBOX_CONFLICT & ~RESPONSE_MAILBOX_CONFLICT.

REQUEST_MAILBOX. A module shall set REQUEST_MAILBOX if ADDR_DECODED and the address corresponds to the module's request mailbox as specified in IEEE Std 896.2-1991. A module shall maintain REQUEST_MAILBOX set until INIT | TRANSACTION_COMPLETE.

RESPONSE_MAILBOX. A module shall set RESPONSE_MAILBOX if ADDR_DECODED and the address corresponds to the modules response mailbox as specified in IEEE Std 896.2-1991. A module shall maintain RESPONSE_MAILBOX set until INIT | TRANSACTION_COMPLETE.

REQUEST_MAILBOX_BUSY. A module shall set REQUEST_MAILBOX_BUSY if the module's request mailbox can not receive the message at the present time, but will be able to if the MASTER retries. A module shall maintain REQUEST_MAILBOX_BUSY set until the module's request mailbox can receive a message.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RESPONSE_MAILBOX_BUSY. A module shall set RESPONSE_MAILBOX_BUSY if the module's response mailbox can not receive the message at the present time, but will be able to if the MASTER retries. A module shall maintain RESPONSE_MAILBOX_BUSY set until-the module's response mailbox can receive a message.

REQUEST_MAILBOX_SELECTED. A module shall set REQUEST_MAILBOX_SELECTED if REQUEST_MAILBOX & (WRITE_UNLOCKED | WRITE_NO_ACKNOWLEDGE) and shall maintain it set until INIT | TRANSACTION_COMPLETE.

RESPONSE_MAILBOX_SELECTED. A module shall set RESPONSE_MAILBOX_SELECTED if RESPONSE_MAILBOX & (WRITE_UNLOCKED | WRITE_NO_ACKNOWLEDGE) and shall maintain it set until INIT | TRANSACTION_COMPLETE.

BROADCAST_SELECT_ATTRIBUTE. A module shall set BROADCAST_SELECT_ATTRIBUTE if BROADCAST and **AD[5..0]*** are released or the message-passing select mask register bit corresponding to the value of **AD[5..0]*** is set. The value of one (1) on **AD[5..0]*** corresponds to the LSB+1 of the register. The value of sixty-three (63) corresponds to the most significant bit of the register. A module shall maintain BROADCAST_SELECT_ATTRIBUTE set until INIT | TRANSACTION_COMPLETE.

REQUEST_MAILBOX_CONFLICT. A module shall set REQUEST_MAILBOX_CONFLICT if the module's request mailbox cannot receive the message during the busy retry interval, but will be able to at a later time. A module shall maintain REQUEST_MAILBOX_CONFLICT set until the module's request mailbox can receive a message.

RESPONSE_MAILBOX_CONFLICT. A module shall set RESPONSE_MAILBOX_CONFLICT if the module's response mailbox cannot receive the message during the busy retry interval, but will be able to at a later time. A module shall maintain RESPONSE_MAILBOX_CONFLICT set until the module's response mailbox can receive a message.

MAILBOX_SELECTED. A module shall set MAILBOX_SELECTED if REQUEST_MAILBOX_SELECTED | RESPONSE_MAILBOX_SELECTED and shall maintain it set until INIT | TRANSACTION_COMPLETE.

MESSAGE_BUSY. A module shall set MESSAGE_BUSY if REQUEST_MAILBOX & REQUEST_MAILBOX_BUSY | RESPONSE_MAILBOX & RESPONSE_MAILBOX_BUSY. A module shall maintain MESSAGE_BUSY set until INIT | TRANSACTION_COMPLETE.

MESSAGE_CONFLICT. A module shall set MESSAGE_CONFLICT if MAILBOX_INITIALIZATION | REQUEST_MAILBOX & REQUEST_MAILBOX_CONFLICT | RESPONSE_MAILBOX & RESPONSE_MAILBOX_CONFLICT. A module shall maintain MESSAGE_CONFLICT set until INIT | TRANSACTION_COMPLETE.

9.2.2 Frame format specification

9.2.2.1 Frame format—non-Futurebus+ message level

The format for a frame not complying with the Futurebus+ message level specification shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to zero (0) or 64–127.
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5) The remainder of the frame may contain user defined data.

9.2.2.2 Frame format—unacknowledged event frame

The format for an unacknowledged event frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to one (1). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to zero (0).
7) The fields consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The remainder of the frame may contain user defined data.

This frame shall be sent to response mailboxes only.

9.2.2.3 Frame format—acknowledged event frame

The format for an acknowledged event frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to two (2). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to a unique identifying number. The sending module shall use a different number for each unique message it is in the process of sending to the same destination module.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The remainder of the frame may contain user defined data.

This frame shall be sent to request mailboxes only.

9.2.2.4 Frame format—event positive acknowledge frame

The format for an event positive acknowledge frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to three (3). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6) Byte 5 shall be set to the message identification received in byte 5 of the acknowledged event frame to which this response applies.
7) The field consisting of bytes 6 and 7 shall be set to eight (8), the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.

This frame shall be sent to response mailboxes only.

9.2.2.5 Frame format—event negative acknowledge frame

The format for an event negative acknowledge frame shall be:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to four (4). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification received in byte 5 of the acknowledged event frame to which this response applies.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The field consisting of bytes 8 through 11 inclusive shall be set to the value of the appropriate exception type as specified in 9.2.6.
9) The remaining bytes in the frame can be used for additional information about the failure.

This frame shall be sent to response mailboxes only.

9.2.2.6 Frame format—multiple frame message request frame

The format for an multiple frame message request frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to five (5). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to a unique identifying number. The sending module shall use a different number for each unique message it is in the process of sending to the same destination module.
7) The field consisting of bytes 6 and 7 shall be set to twelve (12), the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The field consisting of bytes 8 through 11 inclusive shall contain the message size as specified in 9.2.3.

This frame shall be sent to request mailboxes only.

9.2.2.7 Frame format—multiple frame message positive response frame

The format for an multiple frame message positive response frame shall be as follows:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to six (6). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification received in byte 5 of the multiple frame message request frame to which this response applies.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The field consisting of bytes 8 through 11 inclusive shall contain the maximum frame interval as specified in 9.2.4.

This frame shall be sent to response mailboxes only.

9.2.2.8 Frame format—multiple frame message negative response frame

The format for an multiple frame message negative response frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to seven (7). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification received in byte 5 of the multiple frame message request frame to which this response applies.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The field consisting of bytes 8 through 11 inclusive shall be set to the value of the appropriate exception type as specified in 9.2.6
9) The remaining bytes in the frame can be used for additional information about the failure.

This frame shall be sent to response mailboxes only.

9.2.2.9 Frame format—multiple frame message ordered data frame

The format for an multiple frame message ordered data frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) and a WRITE_UNLOCKED transaction shall be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to eight (8). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification sent in byte 5 of the multiple frame message request frame to which this frame applies.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The remainder of the frame contains a fragment of the message.

This frame shall be sent to request mailboxes only.

9.2.2.10 Frame format—multiple frame message sequenced data frame

The format for an multiple frame message sequenced data frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to nine (9). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification sent in byte 5 of the multiple frame message request frame to which this frame applies.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The field consisting of bytes 8 through 11 inclusive shall be a monotonically increasing sequence number. If this is the first sequenced data frame the sequence number shall be one (1).
9) The remainder of the frame may contain a fragment of the message.

This frame shall be sent to request mailboxes only.

9.2.2.11 Frame format—multiple frame message last ordered frame

The format for an multiple frame message last ordered frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) and a WRITE_UNLOCKED transaction shall be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to ten (10). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification sent in byte 5 of the multiple frame message request frame to which this frame applies.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The remainder of the frame may contains the final fragment of the message.

This frame shall be sent to request mailboxes only.

9.2.2.12 Frame format—multiple frame message last sequenced frame

The format for an multiple frame message last sequenced data frame shall be as follows:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to eleven (11). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification sent in byte 5 of the multiple frame message request frame to which this frame applies.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.
8) The field consisting of bytes 8 through 11 inclusive shall be the sequence number of the last sequenced data frame incremented.
9) The remainder of the frame may contain the final fragment of the message.

This frame shall be sent to request mailboxes only.

9.2.2.13 Frame format—multiple frame message positive acknowledge frame

The format for an multiple frame message positive acknowledge frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to twelve (12). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification received in byte 5 of the multiple frame message request frame to which this response applies.
7) The field consisting of bytes 6 and 7 shall be set to eight (8), the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.

This frame shall be sent to response mailboxes only.

9.2.2.14 Frame format—multiple frame message negative acknowledge frame

The format for a multiple frame message negative acknowledge frame shall be as follows:

1) The most significant bit of byte 0 shall be set to zero (0) if a WRITE_UNLOCKED transaction is to be used to transfer the frame. The most significant bit of byte 0 shall be set to one (1) if a WRITE_NO_ACKNOWLEDGE transaction is to be used to transfer the frame.
2) The field consisting of bits 0 through 6 inclusive of byte 0 shall be set to one (1).
3) The field consisting of byte 1 shall be set to the message priority.
4) Bytes 2 and 3 shall be set to the sending module's global identification.
5) The field consisting of bits 0 through 6 inclusive of byte 4 shall be set to thirteen (13). Bit 7 of byte 4 shall be set to specify that an attention request event be sent to the receiving module.
6) Byte 5 shall be set to the message identification received in byte 5 of the multiple frame message request frame to which this response applies.
7) The field consisting of bytes 6 and 7 shall be set equal to the sum of the bytes in the frame. This number shall include all of the bytes in the header and the body of the frame.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8) The field consisting of bytes 8 through 11 inclusive shall be set to the value of the appropriate exception type as specified in 9.2.6.
9) The remaining bytes in the frame can be used for additional information about the failure.

This frame shall be sent to response mailboxes only.

9.2.3 Message size

The message size is the total number of message bytes contained in the fragments of a multiple frame series (ordered frames or sequenced frames).

9.2.4 Frame interval

The frame interval is a positive number measured in units of 2^{-32} s (~233 ps). It indicates the minimum delay between sending any two frames in a multiple frame message.

9.2.5 Sequence number

The sequence number is reserved for use in sequenced multiple frame messages. The sequence numbers for each successive frame in the series will be incremented by one (1).

9.2.6 Exception type field

The following values shall be used for the exception type parameter for negative response frames:

Exception value	Exception description
1	Invalid frame type
2	Invalid frame size
3	Invalid message size
4	Invalid message Id
5	No sequenced frames
6	Timeout
7–63	Reserved
All others	User defined

9.2.7 Protocol definition

9.2.7.1 Master unacknowledged single frame message protocol

To transmit an unacknowledge single frame message, a module shall do the following:

1) Format an unacknowledged event frame as specified in 9.2.2.2.
2) Transmit the frame using the write command specified by the most significant bit of the first byte.

9.2.7.2 Slave unacknowledged single frame message protocol

Upon receiving an unacknowledged single frame message a module shall pass the frame to its application level.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.7.3 Master acknowledged single frame message protocol

To transmit an acknowledged single frame message, a module shall do the following:

1) Format an acknowledged event frame as specified in 9.2.2.3.
2) Transmit the frame using the write command specified by the most significant bit of the first byte.
3) When the module receives an event positive acknowledge frame, or an event negative acknowledge frame, or if a timeout occurs the module shall notify its application level.

9.2.7.4 Slave acknowledged single frame message protocol

After receiving an acknowledged event frame, a module shall do the following:

1) If the frame is successfully received the module shall format an event positive acknowledge frame as specified in 9.2.2.4.
2) If the frame is not successfully received the module shall format an event negative acknowledge frame as specified in 9.2.2.5.
3) The module shall then transmit the frame to the requestor.

9.2.7.5 Master ordered multiple frame message protocol

To transmit an ordered multiple frame message, a module shall do the following:

1) Format a multiple frame message request frame as specified in 9.2.2.6.
2) Transmit the frame using the write command specified by the most significant bit of the first byte.
3) If the module receives a multiple frame message negative response frame or if a timeout occurs the module shall notify its application level and discontinue the sequence.
4) If the module receives a multiple frame message positive response frame it shall format a multiple frame message ordered data frame as specified in 9.2.2.9.
5) The module shall then transmit the frame to the responder after the frame interval has elapsed.
6) The module shall repeat steps 4)–5) until the module has sent all the data except that which will fit in one frame.
7) The module shall then format a multiple frame message last ordered data frame as specified in 9.2.2.11.
8) The module shall then transmit the frame to the responder after the frame interval has elapsed.
9) If the module receives a multiple frame message negative acknowledge frame or if a timeout occurs the module shall notify its application level and discontinue the sequence.
10) If the module receives a multiple frame message positive acknowledge frame it shall notify its application level of the successful completion.

9.2.7.6 Master sequenced multiple frame message protocol

To transmit a sequenced multiple frame message, a module shall do the following:

1) Format a multiple frame message request frame as specified in 9.2.2.6.
2) Transmit the frame using the write command specified by the most significant bit of the first byte.
3) If the module receives a multiple frame message negative response frame or if a timeout occurs the module shall notify its application level and discontinue the sequence.
4) If the module receives a multiple frame message positive response frame it shall format a multiple frame message sequenced data frame as specified in 9.2.2.10.
5) The module shall then transmit the frame to the responder after the frame interval has elapsed.
6) The module shall repeat steps 4)–5) until the module has sent all the data except that which will fit in one frame.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7) The module shall then format a multiple frame message last sequenced data frame as specified in 9.2.2.12.
8) The module shall then transmit the frame to the responder after the frame interval has elapsed.
9) If the module receives a multiple frame message negative acknowledge frame or if a timeout occurs the module shall notify its application level and discontinue the sequence.
10) If the module receives a multiple frame message positive acknowledge frame it shall notify its application level of the successful completion.

9.2.7.7 Slave multiple frame message protocol

After receiving a multiple frame message request frame, a module shall do the following:

1) If the frame is successfully received the module shall format a multiple frame message positive response frame as specified in 9.2.2.13.
2) If the frame is not successfully received the module shall format an multiple frame message negative response frame as specified in 9.2.2.14.
3) The module shall then transmit the frame to the requestor.
4) If the message is a sequenced multiple frame message the module shall then receive the multiple frame message sequenced data frames and the last sequenced frame from the requestor and proceed to 6).
5) If the message is an ordered multiple frame message the module shall then receive the multiple frame message ordered data frames and the last ordered frame from the requestor.
6) If the module does not receive the message correctly it formats a multiple frame message negative acknowledge frame as specified in 9.2.2.8 and transmits it to the requester.
7) If the module receives the message correctly it formats a multiple frame message positive acknowledge frame as specified in 9.2.2.7 and transmits it to the requester.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A
Bibliography

(informative)

[B1] IEC 935 (1990), IEC 935FASTBUS. Modular high speed data acquisition system.

[B2] IEEE P1014.1, Standard for a Futurebus+/VME64 Bridge (Draft No. 2, March 1993).[1]

[B3] IEEE Std 1101.1-1991, IEEE Standard for Mechanical Core Specifications for Microcomputers Using IEC 603-2 Connectors.

[B4] IEEE Std 1101.3-1993, IEEE Mechanical Standard for Conduction-Cooled and Air-Cooled 10SU Modules.

[B5] IEEE Std 1156.1-1993, IEEE Standard for Microcomputer Environment Specifications for Computer Modules.

[B6] IEEE Std 1194.1-1991, IEEE Standard for Electrical Characteristics of Backplane Transceiver Logic (BTL) Interface Circuits (ANSI).

[B7] IEEE Std 1212-1991, IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses.

[B8] IEEE P1212.1-1993, IEEE Standard for Communicating Among Processors and Peripherals Using Shared Memory (Direct Memory Access—DMA) (ANSI).

[B9] IEEE Std 1301-1991, IEEE Standard for a Metric Equipment Practice for Microcomputers—Coordination Document.

[B10] IEEE Std 1301.1-1991, IEEE Standard for a Metric Equipment Practice for Microcomputers—Convection-Cooled with 2 mm Connectors.

[B11] IEEE P1394, High Performance Serial Bus (Draft No. 6.6, Jaunuary 1994).[2]

[B12] IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI) (ANSI).

[B13] Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language,* 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[1]This authorized standards project was not approved by the IEEE Standards Board at the time this went to press, and users should be cautioned that all standards projects are subject to revision prior to approval. The latest draft of this project is available from the IEEE.
[2]See footnote 1.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 35.160

Descriptor: microprocessor

Price based on 208 pages

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale