

This is an **Archive IEEE Standard**. It has been withdrawn or superseded by a later version of the standard.

ARCHIVE STANDARDS MAY HAVE VALUE AS HISTORICAL DOCUMENTS, BUT THEY ARE NOT VALID OR APPROVED IEEE STANDARDS. THE IEEE SHALL NOT BE LIABLE FOR ANY DAMAGES RESULTING FROM THE SUBSCRIBER'S OR USERS' PRACTICE OF THIS UNAPPROVED, ARCHIVE IEEE STANDARD.

For more information about this or other current, revised or withdrawn standards, please consult the IEEE Standards Status report on-line at <http://standards.ieee.org/db/status/>.

IEEE Recommended Practice for Futurebus+

Sponsor
**Bus Architecture Standards Committee
of the
IEEE Computer Society**

Approved October 13, 1993
IEEE Standards Board

Approved April 13, 1994
American National Standards Institute

Abstract: The system layer issues associated with bus-based computer systems are described. It is intended to be a tool for Futurebus+ board and system designers. As such, the intent of this recommended practice is to characterize a variety of systems topics as they relate to Futurebus+. When used in conjunction with other IEEE standards, the details to develop modular, open-architecture-based systems fulfilling user needs across a wide computing spectrum are available. The specifications that may be required in conjunction with this recommended practice are: ISO/IEC 10857: 1994, IEEE Std 896.2-1991, IEEE Std 896.4-1993, IEEE Std 896.5-1993, IEEE Std 1149.1-1990, IEEE Std 1156.1-1993, IEEE Std 1194.1-1991, IEEE Std 1212-1991, IEEE Std 1212.1-1993, IEEE Std 1301.1-1991, and IEEE P1394/D6.8, March 1994.

Keywords: backplane bus, bus, bus architecture, Futurebus+, multiprocessor systems, shared memory 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 1-55937-434-9

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

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

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and service 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 does not mean that using such technology for the purpose of conforming to such standards is authorized by the patent owner. It is the obligation of the user of such technology to obtain all necessary permissions.

Introduction

(This introduction is not part of IEEE Std 896.3-1993, IEEE Recommended Practice for Futurebus+.)

This recommended practice represents the third document in the family of 896 Futurebus+ standards. It is the first Futurebus+ standard sponsored by the IEEE Computer Society Bus Architecture Standards Committee. The initial work in development of the Futurebus+ specification was done under the auspices of the IEEE Computer Society Microprocessor Standards Committee. In 1988 both the Next Generation Computer Resources (NGCR) Task Force of the United States Navy and the VMEbus International Trade Association (VITA) (a trade association of both VMEbus manufacturers and users) agreed to join the IEEE in revising the initial Futurebus+ standard, IEEE Std 896.1-1987. In early 1989 the Multibus Manufacturers Group (MMG) (a trade association of both Multibus I and Multibus II manufacturers and users) also agreed to join the above effort.

The primary goal of all four groups (IEEE, United States Navy, VITA, and MMG) was to provide a new microprocessor bus standard that would be commercially viable and that would be acceptable to the two manufacturer groups and the three user communities.

This work resulted in the IEEE 896 family of standards. IEEE Std 896.1 (now designated ISO/IEC 10857 : 1994¹) defines the logical functionality of the set of signals that make up the bus. IEEE Std 896.2-1991 describes and specifies the physical layer (electrical characteristics, pin-outs, connector locations, board sizes, etc.) requirements. It contains application environment profiles for profiles A, B, and F. IEEE Std 896.3-1993 (this document) describes the system layer of recommended practices (reliability, maintainability, fault management, fault tolerance, serial bus, real-time operation, secure systems, testing, and bus bridges) associated with bus-based computer systems. It is intended to be a tool for Futurebus+ board and system designers. As such, the sections included in this recommended practice are not written in a specification format, but rather as a discussion of diverse topics as they relate to Futurebus+.

Where deemed appropriate by the working group and the sponsoring organization, further sections on major recommended practices may be added to this recommended practice in a supplement.

The software interface to the core registers, which are used to reset and configure nodes on a bus, is defined in IEEE Std 1212-1991. This definition is also used in IEEE Std 1596-1992 and IEEE P1394/D6 .8, March, 1994.

This recommended practice, while being one of the more conceptual of the Futurebus+ family, has been produced by an extremely dedicated team of editors. These individuals are both significant experts in their fields as well as hard working and committed to strict, yet self-imposed, deadlines.

Recognition for their hard work and dedication goes to all who have donated their time and talents to the development of this recommended practice: Mark Bunker, for his experience with the bus and system-related issues surrounding the initialization; Bob Greiner, for his intuition and creativity in developing the byte lane mapping model for Futurebus+; Mike Foster, for his expertise with fault management, fault tolerance, and reliability; Mike Dorsett, for his contributions to the topics of reliability and maintainability; Sam Duncan, for his insight into fault management and systems test; George Nacht, for his experience with techniques for performance measurement; Richard Fryer, for his contributions into real-time nonintrusive instrumentation; Don Senzig, for his thoughtful insight into the uses of a serial bus in a system, and for his contributions to bus and system initialization; Jim Leahy, for his expertise in building bus bridges; Gary Reid and Tim Iacono, for their contributions on dual bus architectures in fault-tolerant systems; Lui Sha, Ragunathan Rajkumar, and David Kirk, for their comprehensive look into the issues surrounding priority-based scheduling for real-time systems; Brian Tretick, for his characterization of security-related issues; and Bill Evertz, for his contribution on topics related to arbitrating for a bus. These are the contributors to the sections of this recommended practice. They were helped by other volunteers too numerous to mention individually, but whose assistance was necessary to achieve the quality and comprehensiveness that is in this recommended practice. Finally as the draft editor, it was the responsibility of Thanos Mentzelopoulos to assure that the individual sections were integrated into a consistent document.

¹Information on references can be found in section .2.

At the time this recommended practice was completed, the Futurebus+ Working Group had the following membership:

Ed Jacques, Chair

Barbara Aichinger	Richard Fryer	Ragunathan Rajkumar
Ray Alderman	Bob Greiner	Gary Reid
Jeff Beachy	David Gustavson	Michael Roby
Harrison Beasley	Mark Hassel	Don Senzig
Andy Bonaffini	John Hyde	Lui Sha
Paul Borrill	Tim Iacono	Patricia Smith
Marc Briel	David James	Mike Snodgrass
Mark Bunker	Anatol Kaganovich	Richard Spratt
Paul Cook	David Kirk	Dan Tarrant
Jon Curry	Shigeo Kobayashi	Lars Thernsjo
Sudhir Dhawan	Ralph Lachenmaier	John Theus
Paul Dixon	Jim Leahy	Joe Trainor
Mike Dorsett	Kent Leung	Brian Tretick
Frank Duffy	Alexander Lewin	Larry Tyson
Sam Duncan	Thanos Mentzelopoulos	LCDR Robert Voigt
Bruce Dunlop	Jim Moidel	Bob Widlicka
Mike Ebl	George Nacht	Dwight Wilcox
Bill Evertz	Eric Neely	Mark Williams
Wayne Fisher	Thomas Patrick	Joel Witt
Mike Foster	Martin Pullam	David Wright

The following persons were on the balloting committee:

Charles Abernethy	Robert Crowder	A. D. Heher
Amir Abouelnaga	Jonathan Crowell	Hans Heilborn
William Adams	Rob Davies	Frank Hom
Barbara Aichinger	Stephen Deiss	Zoltan Hunor
Edward Aichinger	Dante Del Corso	Peter Ilieve
Scott Akers	Ian Dobson	Ed Jacques
Ray S. Alderman	Michael Dowling	David James
Richard P. Ames	Jean-Jacques Dumont	Norman Joehlin
Duane Anderson	Samuel Duncan	Jack Johnson
Richard Andreano	Sourav Dutta	Horace Jones
Keith Anthony	Christopher Eck	Daavid Kaminski
Bjorn Bakka	R. D. Edwards	Richard Karpinski
Harrison Beasley	Wilhelm P. Evertz	Michael Khvilivitzky
Russell Beverly	Wayne Fischer	Frederick Ki
Kyle Black	Chee K. Fong	Larry Kinney
Paul Borrill	Gordon Force	Christopher Koehle
David Brearley	Ernest Franke	Ernst kristiansen
John Brightwell	Andrew Fraser	Ralph Lachenmeir
Scott Buck	Joseph George	Lak Ming Lam
Fletcher Buckley	Isaac Ghansah	Tuvia Lamdan
Clyde Camp	Larry Gilbert	Min-Suk-Lee
George Carson	Julio Gonzalez-Sanz	Michael Lee
C. H. Chen	John Griffith	Boon Lum Lim
Kim Clohessy	Mark Hassel	Rollins Linser
Steven Cobb	David Hawley	Joe Macura
Paul D. Cook	Michael Hayward	C. Mallipreddi
Antonio Corradi	Kenneth Heck	Tulin Mangir

Arthur Mansky
 Joel Martinez
 William McDonald
 Thanos Mentzelopoulos
 Milan Melenkovic
 Brace Millard
 Marlyn Miner
 James Moloney
 William Molyneaux
 Brian Morrison
 Klaus Mueller
 George Nacht
 Mitsunori Nakata
 J. D. Nicoud
 Tadahiko Nishimukai
 Gregory Novak
 P. Onvfyk
 Michael Orlovsky
 Francisco Pataro
 Chandresh Patel
 L. Patnaik
 Mira Pauker
 Donald Pavlovich

Philip Piele
 Shlomo Pri-Tal
 Rochit Rajsuman
 Hira Ranga
 Richard Rawson
 William Rhoades
 Bill Rigby
 Gary Robinson
 Fred Rosenberger
 Debabrata Sarma
 Frederick Sauer
 Bennett Scharf
 Norman Schneidewind
 Rudolf Schubert
 Don Senzig
 Kerry Shore
 F. P. Smerke
 Patricia Smith
 F. Gerald Snyder
 Robert Stewart
 Fred Strauss
 Dongzhuang Su
 Hao Tang

Daniel Tarrant
 Manu Thapar
 Michael Thompson
 Robert Tripi
 Robert Voigt
 Yoshiaki Wakimura
 Carl Warren
 Clarence Weaver
 Michael Wenzel
 Dave Wickliff
 Robert Widlicka
 Dwight Wilcox
 James Wilkinson
 Andrew Wilson
 Gregory Winston
 John Wise
 Joel Witt
 Ronald Woan
 David Wright
 Oren Yuen
 Janusz Zalewski

The final conditions for approval of this standard were met on October 13, 1993. This standard was conditionally approved by the IEEE Standards Board on September 15, 1993, with 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

Rachel A. Meisel
IEEE Standards Project Editor

CLAUSE	PAGE
1. Overview.....	1
1.1 Scope.....	1
1.2 Terminology.....	2
2. References.....	2
3. Definitions.....	3
3.1 Definitions for bus lines and signals.....	3
3.2 General definitions.....	3
3.3 Futurebus+ logo.....	11
4. Bus line description.....	11
4.1 Information lines.....	13
4.2 Synchronization lines.....	14
4.3 Arbitrated message lines.....	15
4.4 RE* Reset/Bus initialize.....	15
4.5 Central arbiter.....	15
5. System/Bus initialization.....	16
5.1 Introduction.....	16
5.2 System initialization and operation summary.....	16
5.3 Bus initialization.....	19
6. Byte lane mapping.....	26
6.1 Introduction.....	26
6.2 Endian history.....	26
6.3 Addressan.....	29
6.4 Futurebus+ addressan standard.....	30
6.5 Relabeling.....	31
6.6 Multiplexers.....	32
7. Fault management.....	34
7.1 Introduction.....	34
7.2 System fault management.....	36
8. Reliability and maintainability.....	67
8.1 Introduction.....	67
8.2 Reliability theory.....	68
8.3 Part selection and control.....	68
8.4 Derating.....	68
8.5 Environmental design.....	70
8.6 Failure modes and effects analysis (FMEA).....	70
8.7 EMI/RFI and ESD shielding.....	74
8.8 Arbitration.....	75
8.9 Live insertion.....	76

CLAUSE	PAGE
8.10 Serial bus	76
8.11 Redundancy.....	76
8.12 Test architecture.....	77
8.13 Conformance testing	77
9. System test	79
9.1 Introduction.....	79
9.2 Diagnostics and open systems.....	79
9.3 Caused-All-Faults testing.....	99
9.4 System verification/validation	102
10. Serial bus.....	102
10.1 Introduction.....	102
10.2 Requirements	102
10.3 Node design.....	112
10.4 Serial bus specification	113
11. Futurebus+ Profile B bridge.....	115
11.1 Scope.....	115
11.2 Description	115
12. Dual bus architectures	129
12.1 Introduction.....	129
12.2 Backplane signals.....	129
12.3 Operational modes	129
12.4 Power supplies	135
13. Real-time systems	138
13.1 Introduction.....	138
13.2 Priority-driven scheduling overview.....	139
13.3 Principles of priority management for real-time systems	142
13.4 Real-time scheduling support on the Futurebus+	148
13.5 Clock management.....	154
13.6 Predictable cache performance in real-time systems	161
13.7 Software configuration guide.....	164
14. Secure systems	165
14.1 Introduction.....	165
14.2 Approach.....	166
14.3 Security design issues	171
14.4 Security service implementation.....	186
15. Arbitration.....	192
15.1 Introduction.....	192
15.2 Distributed arbitration	193

CLAUSE	PAGE
15.3 Central arbitration	198
15.4 Split transactions	207
15.5 Physical aspects of central arbitration.....	208
16. Measurements	209
16.1 Introduction.....	209
16.2 Data samples	211
16.3 Trace data source	214
16.4 Sampling	217
16.5 Filter	218
16.6 Sample data collection	218
Annex A (Informative) Scheduling real-time tasks	220
Annex B (Informative) Bibliography.....	230

IEEE Recommended Practice for Futurebus+

1. Overview

1.1 Scope

Futurebus+ provides a rich set of features that, when used properly, allows the designer to construct systems for a broad range of applications. Futurebus+ can be applied to general purpose systems whose sizes range from mainframes to the desktop. In addition, Futurebus+ has unique features that are expected to support special purpose applications such as real-time control, high-reliability, high-availability systems, and secure computing.

This document describes recommended practices for systems built to the Futurebus+ family of standards. The sections included in this recommended practice are broken into two categories—general design practices and specific practices.

General design practice sections are included for the following topics:

- a) System and bus initialization
- b) Byte lane mapping
- c) Fault management
- d) Reliability and maintainability
- e) Systems test
- f) Serial bus
- g) Bus bridge
- h) Arbitration

Specific practice sections are provided for the following system types:

- a) Fault tolerant
- b) Real time
- c) Secure computing

In each section descriptions are provided for the intended use of facilities that exist in ISO/IEC 10857 : 1994, IEEE Std 896.2-1991, and IEEE Std 1212-1991.¹ In addition, common terms and definitions are provided to facilitate a common framework for discussion and reference to concepts that are relevant to the topic of discussion.

All features mentioned in this recommended practice are optional for Futurebus+ systems, unless they are explicitly required by a Futurebus+ profile.

1.2 Terminology

The word *may* indicates flexibility of choice with no implied preference.

The word *shall* indicates a mandatory requirement. Designers shall implement all such mandatory requirements in order to ensure interoperability with Futurebus+ conformant products and claim conformance to the Futurebus+ standards.

The word *should* indicates flexibility of choice with a strongly preferred implementation. The phrase *it is recommended* is used interchangeably with the word *should*.

2. References

The following publications should, when applicable, be used in conjunction with this recommended practice:

EIA IS-64, Two Millimeter, Two-Part Connectors for Use with Printed Boards and Backplanes, March, 1991.²

IEEE Std 896.2-1991, IEEE Standard for Futurebus+—physical Layer and Profile Specification (ANSI).³

IEEE Std 896.2a-1994, IEEE Standard for Errata, Correction and Clarification of IEEE Std 896.2-1991, IEEE Standard for Futurebus+—Physical Layer and Profile Specification

IEEE Std 896.4-1993, IEEE Standard for Conformance Test Requirements for Futurebus+.

IEEE Std 896.5-1993, IEEE Standard for Futurebus+, Profile M (Military) (ANSI).

IEEE Std 896.9/D2.5R May 1994, Standard for Fault Tolerance Extension to the Futurebus+ Architecture.⁴

IEEE Std 1014-1987, IEEE Standard for a Versatile Backplane Bus: VMEbus (ANSI).

IEEE Std 1101.2-1992, IEEE Standard for Mechanical Core Specifications for Conductor-Cooled Eurocards (ANSI).

IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture.

IEEE Std 1156.1-1993, IEEE Standard for Microcomputer Environment Specifications for Computer Modules.

IEEE Std 1194.1-1991, IEEE Standard for Electrical Characteristics of Backplane Transceiver Logic (BTL) Interface Circuits (ANSI).

¹Information on references can be found in section 2.

²EIA publications are available from Global Engineering, 1990 M Street NW, Suite 400, Washington, DC, 20036, USA.

³IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

⁴At the time this went to press, IEEE P896.9/D2.5R, May 1994 was not approved by the IEEE Standards Board. It is available from the IEEE.

IEEE Std 1196-1987, IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus® (ANSI).

IEEE Std 1212-1991, [ISO/IEC DIS 13121] IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Bus (ANSI).

IEEE Std 1275-1994, IEEE Standard for Boot Firmware: Core Requirements and Practices.⁵

IEEE P1394/D6.8 March 1994, High Performance Serial Bus.⁶

ISO/IEC 10857: 1994, [ANSI/IEEE Std 896.1 1994 Edition], Information technology—Microprocessor systems—Futurebus+—Logical protocol specification.⁷

ISO/IEC 10861: 1994, [ANSI/IEEE Std 1296 1994 Edition], Information technology—Microprocessor systems—High-performance synchronous 32-bit bus: MULTIBUS II.

3. Definitions

This section includes definitions related to the description of Futurebus+ and Futurebus+ systems. Some of the definitions included in this recommended practice differ from those in other IEEE Standard 896 documents. This was done in order to ensure compliance with other IEEE standards as official policies.

3.1 Definitions for bus lines and signals

The following words define the use of terms that are specific to Futurebus+ signals:

3.1.1 activate: To apply signals to one or more bus lines. Similarly, the term *activated* is used to describe the state of a group of bus lines when they carry signals.

3.1.2 assert: To apply a logic one signal to a bus line. Similarly, the term *asserted* is used to describe the state of a bus line when the signal it carries represents a logic one.

3.1.3 bus line: Signal transmission line, that may be driven by several modules simultaneously using drivers with wire-OR capability. Therefore, a signal carried by a bus line is the combination of signals applied to that line from each module.

3.1.4 release: To apply a logic zero signal to a bus line. Similarly, the term *released* is used to describe the state of a bus line when the signal it carries represents a logic zero.

3.2 General definitions

The following words define terms as they relate to the topics discussed in this recommended practice:

3.2.1 address invariance: When a multi-byte data item is transferred over the bus, bytes with the same relative memory address are always mapped to the same bus lanes, regardless whether the processor is big or little endian and regardless of the significance of the bytes.

3.2.2 address only transactions: Bus transactions that do not include a data phase. The only information transferred is contained within the connection phase and, in some cases, the disconnection phase.

⁵As this standard goes to press, IEEE Std 1275-1994 is approved but not yet published. The draft standard is available from the IEEE. Anticipated publication date is September 1994. Contact the IEEE Standards Department at 1 (908) 562-3800 for status information.

⁶At the time this went to press, IEEE P1394/D6.8, March 1994 was not approved by the IEEE Standards Board. It is available from the IEEE.

⁷This standard is available from the IEC, Case Postale 131, 3 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse and from the IEEE.

3.2.3 administrative security: Management constraints, operational procedures, and other administrative controls to enforce a security policy.

3.2.4 alternate access unit (AAU): Type of unit architecture that defines access between multiple IEEE Standard-compliant buses (e.g., Futurebus+ and Serial Bus) when the buses share a common module.

3.2.5 aperiodic tasks: Tasks that arrive at irregular intervals and have only soft (not rigid) deadlines, but a good response time is typically desirable.

3.2.6 application environment profile (AEP): A document that describes functional requirements and points to existing standards, selecting and binding options within those standards. An implementor who then designs a specific module and/or system should be reasonably assured that another designer's (manufacturer's or supplier's) modules will properly function within the same system. This includes all aspects of definition: mechanical, electrical, protocol, environmental, and system considerations.

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

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

3.2.9 atomic transaction: A transaction is atomic (or indivisible) if the only system states on which other transactions can act (which are visible to other transactions) are states prior to the commencement of the transaction and states after the completion of the transaction.

3.2.10 availability: The probability that a system will be able to execute a function accurately at any given time.

3.2.11 backplane: Electronic circuit board and connectors used to interconnect modules together electrically. The backplane connects selected pins of the connectors, thus providing the medium for the transfer of signals needed for the operation of the bus.

3.2.12 backplane bus: A means to connect circuit modules using common signal traces on a backplane and a standard set of rules.

3.2.13 barrier transaction: Transaction that is guaranteed to become visible to other observers after all transactions created before it have become visible.

3.2.14 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.

3.2.15 big addressan: Bus that multiplexes the most significant byte of the address with the data byte that has the lowest address.

3.2.16 big endian: The most significant byte of a data item, it has the lowest relative memory address. Correspondingly, less significant bytes have higher relative memory addresses.

3.2.17 block copy: A series of read or write transactions to sequential memory locations.

3.2.18 board: A physical component that is inserted into one of the backplane slots. Note that a board may contain two nodes.

3.2.19 browsing: Attempts by a user or intruder to access information to which read access is not authorized or intended. Browsing includes the threat of inadvertent access to sensitive information by users and non-users (e.g., over displays visible to others, hardcopy output at printers, misrouted electronic mail). Browsing could violate the principle of least privilege, need-to-know requirements, or clearance authorizations and could result in the unauthorized disclosure of sensitive or classified information.

3.2.20 bus-bridge: 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 bridge from Futurebus+ to VMEbus or to MULTIBUS II).

3.2.21 bus interface unit (BIU): The logic on a module that converts bus signals to and from signals that are compatible with the functional logic of the module.

3.2.22 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.

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

3.2.24 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.

3.2.25 byte: A set of eight adjacent binary digits.

3.2.26 byte lane: A data path formed by eight data lines and one parity line, used to carry a single byte among the system modules.

3.2.27 cache: A buffer inserted between one or more processors and the bus, used to hold currently active copies of blocks from main memory.

3.2.28 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.

3.2.29 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.

3.2.30 communication access: Passive and active attacks on information transmitted over communication channels and on the system's communication services themselves. This threat area assumes that an intruder has access to the communication media or the components (both hardware and software) that provide the communication services.

3.2.31 communication security: Protective measures for information transmitted between system components, over telecommunication links, and through networks to provide data confidentiality, integrity, and authenticity.

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

3.2.33 component failure: Malfunctions in the system hardware (e.g., failures in equipment, electronics, input/output device, distribution system), software (e.g., due to deadlock conditions, exceptions, error conditions), or media that are not precipitated through design flaws. Component failures could be manifested because of faulty equipment, unanticipated system events, or environmental effects (e.g., power surge, humidity, heat), and could result in denial of service conditions. However, failure of components that implement security mechanisms could result in the violation of the system security policy through unauthorized disclosure or modification of information, unauthorized receipt of services, or denial of service to legitimate users or critical functions.

3.2.34 computer security: Protection of information, system resources, and system services through controls provided by hardware and software mechanisms, including access controls, user authentication mechanisms, and audit facilities.

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

3.2.36 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.

3.2.37 control acquisition: The total of all bus activity associated with acquiring exclusive control of the bus by a module.

3.2.38 CSR: Control and Status Register.

3.2.39 CSR architecture: Control and Status Register Architecture (see IEEE Std 1212-1991).

3.2.40 CSR unit architecture: The logical component of a node that is accessed by I/O driver software. After the node is initialized and configured, the units normally operate independently. Note that one node could have multiple units (for example, processor, memory, and SCSI controller).

3.2.41 damage: Attacks against system components (e.g., hardware, distribution system, media, software) to destroy the component or render it useless. Damage could be perpetrated inadvertently or intentionally using physical, electrical, or environmental (e.g., fire, water exposure) means, and could result in denial of service conditions.

3.2.42 data encryption standard (DES): A private key cryptography standard promoted by the United States government for use with private or unclassified information.

3.2.43 data invariance: A technique for mapping together big endian and little endian domains by mapping together bits with the same significance. This shall be done with some reference data width, typically either 32 bits or the implementation-dependent local data bus width. Futurebus+ is address invariant, but not necessarily data invariant.

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

3.2.45 deadlock: A situation in which computer processing is suspended when modules are awaiting actions that can only be performed by those waiting, and those waiting cannot perform actions.

3.2.46 denial of message service: Preventing or delaying the performance of legitimate or critical communication services. Denial of message service attacks may be perpetrated at any point in the communication architecture (e.g., data link, network, transport, application), and could result in denial of service conditions.

3.2.47 direct memory access (DMA): Ability of I/O controller modules to independently access memory. An I/O controller with DMA capabilities can access commands, fetch data, and report status by accessing memory directly.

3.2.48 disconnection phase: Period at the end of 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.

3.2.49 doublet: A set of adjacent bytes.

3.2.50 dual buses: Two functional buses (primary and secondary) on the same backplane, each possessing the signals required to transfer data between boards.

3.2.51 electromagnetic interference (EMI): Signals emanating from external sources (e.g., power supplies and transmitters) or internal sources (e.g., adjacent electronic components, energy sources) that disrupt or prevent operation of electronic systems. *See also:* **emanations security**.

3.2.52 emanations attacks: Passive collection, analysis, and interpretation of electromagnetic emissions from and the direction of electromagnetic interference at electrical and electronic equipment. Emanations attacks may be perpetrated at several points in a system (e.g., processors, input/output devices, communication channels, power sources), and could result in unauthorized disclosure or modification of information, denial of service to legitimate users or critical functions, or destruction of sensitive electronic components (e.g., from an electromagnetic pulse created by a nuclear explosion).

3.2.53 emanations security: Protection of electromagnetic emanations of systems from unauthorized interception and analysis, and protection of the system from electromagnetic interference. *See also:* **tempest** and **electromagnetic interference**.

3.2.54 error: Manifestation of a fault within a system (hardware or software).

3.2.55 error detection: Observation of an error by an error detector.

3.2.56 error log: Memory space specifically allocated for recording errors.

3.2.57 error report: A complete or partial summary of the error log.

3.2.58 error strategy: Methodology targeted at dealing with temporary errors.

3.2.59 exerciser: Physical activity generator designed to manifest defects as errors.

3.2.60 failure: The inability of a system or component to perform its required functions within specified performance requirements.

3.2.61 fault: Erroneous hardware or software state resulting from component failure, operator error, physical interference from the environment, design error, program error, or data structure error.

3.2.62 fault coverage: Quality metric used to measure diagnostic tests or automatic test equipment.

3.2.63 fault, intermittent: A recurring temporary error caused by component degradation or inadequate design (e.g. noise margin).

3.2.64 fault, permanent: A continuous and stable failure or error.

3.2.65 fault tolerance: Ability of a system or component to continue normal operation despite the presence of hardware or software faults.

3.2.66 fault, transient: A nonrecurring temporary error caused by temporary environmental conditions.

3.2.67 federal information processing standard publication (FIPS PUB XXXX): Issued by the National Institute of Standards and Technology and available from the National Technical Information Service.

3.2.68 field replaceable unit (FRU): The smallest subassembly that can be swapped in the field to repair a fault.

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

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

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

3.2.72 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.

3.2.73 JTAG: Test access port and boundary scan architecture (see IEEE Std 1149.1-1990).

3.2.74 little addressan: Bus that multiplexes the least significant byte of the address with the data byte that has the lowest address.

3.2.75 little endian: The least significant byte of a data item, it has the lowest relative address. Correspondingly, more significant bytes have higher relative memory addresses.

3.2.76 live insertion: Process of inserting boards into or withdrawing boards from a backplane when power is on. Insertion of boards into a backplane can be performed when power is off or when power is on.

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

3.2.78 locking: 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.

3.2.79 maintainability: The measure of the ability of an item to be retained in, or restored to, a specified condition when maintenance is performed by personnel having specified skill levels, using prescribed procedures and resources, at each prescribed level of maintenance and repair.

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

3.2.81 master elect: Module that has won the most recent arbitration competition.

3.2.82 message stream modification: Attempts to modify, delete, reorder, duplicate, or insert information while the message stream is being transmitted over a communication channel. Message stream modification attacks may be

perpetrated at any point in the communication architecture (e.g., data link, network, transport, application), and could result in unauthorized modification of information or unauthorized receipt of services.

3.2.83 misuse: Use of processing or communication services for other than official or authorized purposes (e.g., personal gain, espionage). Misuse includes the threats of inadvertent or intentional execution of malicious functions (e.g., computer virus, Trojan horse), performance of undesirable functions (e.g., erasing the file system), and general perpetration of errors of commission, omission, and oversight. Misuse could result in unauthorized disclosure or modification of information, unauthorized receipt of services, or denial of service to legitimate users or critical functions.

3.2.84 module: Collection of circuitry designed to perform specific functions that includes an interface to Futurebus+.

3.2.85 monarch processor: The processor selected to manage the configuration and initialization of all modules on one logical bus.

3.2.86 multi-level security: The capability of simultaneously separating and protecting information of two or more security levels during processing.

3.2.87 NIST: National Institute of Standards and Technology.

3.2.88 node: A set of control register addresses (including identification-ROM and reset command registers), that are initially defined in a 4 kb (minimum) initial node address space. Each node can be reset independently (a reset of one node has no affect on other nodes).

3.2.89 node_id: A 16-bit number that defines the initial node address space. During system initialization, systemwide unique node_id values may be assigned to nodes within a tightly coupled system.

3.2.90 NSA: National Security Agency.

3.2.91 object: A passive entity in a system that contains or receives information. Typically, objects include files, directories, registers, buffers, cache, memories, bus lines, displays, and other input/output devices.

3.2.92 octlet: A set of eight adjacent bytes.

3.2.93 page (or segment): The smallest portion of memory that can be relocated by memory mapping techniques.

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

3.2.95 penetration: Attacks by unauthorized persons in attempts to gain system access by defeating the system security perimeter (e.g., log-in controls, access controls, bypass controls). Penetration often is in conjunction with browsing or misuse, and could result in unauthorized disclosure or modification of information, unauthorized receipt of services, or denial of service to legitimate users or critical functions.

3.2.96 periodic tasks: Tasks that have to be processed at regular intervals, and each instance shall be normally completed before the next instance of the same task arrives.

3.2.97 personnel security: Procedures to ensure that personnel with access to sensitive information and critical services have the appropriate authorizations and training.

3.2.98 physical security: Protection of system resources from physical access, tampering, and destruction, such as through the use of barriers, locks, seals, and intrusion detection systems.

3.2.99 priority inheritance: This occurs when a lower priority message at the head of the queue uses the priority of a higher priority message that is blocked.

3.2.100 priority inversion: Condition that can occur when a higher priority activity arrives while a lower priority activity is using the shared resource, and the higher priority activity cannot preempt the lower priority activity.

3.2.101 preemption: Process in which the current bus master relinquishes the bus because another module has requested it. In some systems any module may cause preemption; in some systems only a module with a higher priority request may cause preemption.

3.2.102 primary bus: The collection of signals that provides the system with the basic mechanism for exchanging data between boards.

3.2.103 quadlet: A set of four adjacent bytes.

3.2.104 quadrant: This pertains to the investigation and study of tamper protection concepts and mechanisms. *See also: tamper protection.*

3.2.105 real-time system: A system in which the correctness of a computation depends not only upon the results of the computations but also upon the time at which the outputs are generated.

3.2.106 reliability: The probability that an item will perform its intended function for a specified interval under stated conditions.

3.2.107 request: Transaction that is 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.

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

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

3.2.110 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.

3.2.111 risk: The potential for loss (such as the compromising of data confidentiality or data integrity or the denial of service to users) that could result from threats to the system, exploiting vulnerabilities in the system. *See also: threat.*

3.2.112 rogue module: An unauthorized module introduced into the system to perform malicious activities, or an authorized module corrupted by malicious hardware or software.

3.2.113 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 requesting the bus at the same priority level have had control of the bus.

3.2.114 secondary bus: A collection of signals that provides the system an alternate mechanism for exchanging data between boards as a means to recover from faults in the primary bus.

3.2.115 security level: The sensitivity of information represented, for example, by a combination of hierarchical classifications and nonhierarchical categories.

3.2.116 security policy: The objectives and mandates for protecting information, services, and other resources in a system, and the philosophy of protection for meeting those objectives.

3.2.117 security service: The capability of the system to ensure the security of system resources or data transfers. Access controls, authentication, data confidentiality, data integrity, and nonrepudiation are traditional data communications security services.

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

3.2.119 sensitivity label: Representation of the security level of an object describing the sensitivity (e.g., classification) of information in the object.

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

3.2.121 single-level security: The capability of protecting only one security level during processing.

3.2.122 single point of failure: The part of a system whose failure would result in the inability of the system to perform its intended function.

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

3.2.124 snarf: This occurs when a module takes a copy of data passing by on the bus even though it did not request it.

3.2.125 snoop: To monitor the transaction by a module that is not the master that originated the transaction or the repository of last resort for the data. (Cache memories snoop transactions to maintain coherence.)

3.2.126 split transaction: An operation in which the request is transmitted in one bus transaction and the response is transmitted in a separate subsequent bus transaction.

3.2.127 sporadic tasks: Asynchronous tasks that may have hard (rigid) deadlines that must be met, but have a minimum interarrival duration between instances (typically by enforcement).

3.2.128 spurious initiation: Attempts to establish a communication connection or association, using a false identity or through the replay of a previous, legitimate initiation sequence. Spurious initiation includes spoofing or masquerading attempts in the communication system, and coupled with other attacks, could result in unauthorized disclosure or modification of information, unauthorized receipt of services, or denial of service to legitimate users or critical functions.

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

3.2.130 subject: An active entity in a system that causes information to flow between objects or otherwise affects the system state. Typically, subjects include users, processes, and processing elements.

3.2.131 substitution: The introduction unauthorized, potentially malicious components to intercept communications, generate incorrect or misleading information, masquerade as a legitimate component, or perform other undesirable functions. Substitution may be perpetrated throughout a system (e.g., hardware and software components), and could result in unauthorized disclosure or modification of information, unauthorized receipt of services, or denial of service to legitimate users or critical functions.

3.2.132 system access: Attacks on computer system resources (e.g., files, directories, devices) from an intruder having access to the application software or operating system. This threat area assumes that an intruder has access to some form of man-machine interface in the system (e.g., terminal, operator console, workstation).

3.2.133 system failure: Malfunctions in the hardware and software that could compromise the security of the system (nonsecurity-related failures and design flaws are not considered). The malfunctions include both intentional and inadvertent design or implementation flaws (including malicious hardware and software) and component failures. For intentional attacks, this threat area assumes that an intruder has access to the system's design or implementation processes or to the operational system in such a way as to be able to cause a failure in a component. For inadvertent attacks, this threat area assumes that there may not be a specific intruder.

3.2.134 system high: The highest security level supported by a system.

3.2.135 system low: The lowest security level supported by a system.

3.2.136 tampering: Attacks to physical and logical system components (i.e., hardware, software, media), including substituting rogue components in place of legitimate components, physical theft of components, damage to components, and electrical probing within the system. This threat area assumes that an intruder has physical or electrical access to the system components and their internal structure (e.g., processor and memory boards, software modules).

3.2.137 tamper protection: The aspect of physical security concerned with the protection of system resources from unauthorized modification, especially where modifying or altering a component degrades system security. *See also:* **quadrant**.

3.2.138 tempest: The investigation, study, and control of spurious electromagnetic signals emitted by electronic equipment. *See also:* **emanations security**.

3.2.139 temporary: Intermittent or transient.

3.2.140 test: An activity in which a system or component is operated under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

3.2.141 theft: Unauthorized removal of system components (i.e., hardware, software, media). Theft could result in unauthorized disclosure of information or sensitive technology and denial of service conditions.

3.2.142 threat: Means by which a system may be adversely affected. Threats include both inadvertent and malicious actions.

3.2.143 traffic analysis: The collection, analysis, and interpretation of communication patterns to infer operational and logistics-related information. Traffic analysis may be perpetrated using the same techniques as wiretapping, and could result in the unauthorized disclosure of sensitive operations or logistics information.

3.2.144 trusted computing base (TCB): The totality of security controls and mechanisms within an information system.

3.2.145 wiretapping: Passive surveillance of communication channels to gain access to information transmitted over those channels. Wiretapping may be perpetrated through physical, electrical, and radio-frequency taps into the communication channel, and could result in the unauthorized disclosure of information transmitted over communication channels. Also known as eavesdropping.

3.2.146 vendor-defined: Parameters that identify and particularize a specific vendor implementation. Although IEEE Std 1212-1991, CSR Architecture, specifies the size, location, and a few values within fields, their remaining values are provided by the module vendor. *Syn:* **vendor-dependent.**

3.3 Futurebus+ logo

The 896 logo is copyrighted by the IEEE Computer Society, and shall only be used on products that fully conform to all mandatory requirements of an IEEE approved 896 profile document or adhere to IEEE Std 896.1, or IEEE Std 896-1987. Figure 1 shows the P896 logo.



Figure 1—896 logo

4. Bus line description

Table 1 shows the list of bus lines, while subclauses 4.1 through 4.5 give brief descriptions of their function. The signals that use these lines are more fully described in later sections.

Table 1— List of bus lines

Data path width		32-bit	64-bit	28-bit	256-bit
<i>Information lines</i>					
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
<i>Synchronization lines</i>					
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
<i>Arbitrated message lines</i>					
AB[7..0]*	Arbitrated Message Bus	8	8	8	8
ABP*	Arbitrated Message Bus Parity	1	1	1	1
AP*,AQ*,AR*	Arbitrated Message Sync	3	3	3	3
AC[1..0]*	Arbitrated Message Condition	2	2	2	2
<i>Reset/Bus Initialize line</i>					
RE*	REset	1	1	1	1
<i>Central arbitration</i>					
RQ[1..0]*	ReQuest	2	2	2	2
GR*	GRant	1	1	1	1
PE*	PrEmption	1	1	1	1
GA[4..0]*	Geographical Address	5	5	5	5
Total active signals		96	132	204	348

When appended to a signal's name, the suffix * (asterisk) 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.

Lines within a group of bus lines are numbered by the same letters, e.g., **AD0***, **AD1***, **AD2***, etc. Notation **AD[63..0]*** is used to represent a group of lines or signals. Notation **AD[]*** is used to refer to all of the lines within the group.

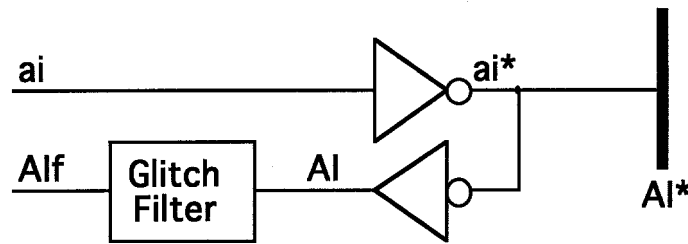


Figure 2— Signal conventions

Figure 2 shows an example of the signal conventions used in Futurebus+. The example is shown with open collector technology (although Futurebus+ 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 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***.

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

4.1 Information lines

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

4.1.1 AD[63..0]* Address/Data lines

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 slave or intervening slave during read transactions. During the first data beat of partial transfers **AD[63..0]*** carry lane deselect signals. During cached split transactions **AD[63..0]*** carry the requesting module's address and priority and the responding module's status. During noncached split transactions **AD[63..0]*** carry the requesting modules global identification and the transaction identification.

4.1.2 D[255..64]* Data lines

These bidirectional data lines carry 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.

4.1.3 CM[7..0]*

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

4.1.4 CP* Command parity

The line that carries the parity for the command lines.

4.1.5 ST[7..0]* Status

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

4.1.6 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.

4.1.7 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.

4.1.8 TG[7..0]* Tag

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

4.1.9 TP* Tag parity

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

4.2 Synchronization lines

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

4.2.1 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.

4.2.2 AK* Address acknowledge

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

4.2.3 AI* Address acknowledge inverse

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

4.2.4 DS* Data synchronization

A synchronization signal that is 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.

4.2.5 DK* Data acknowledge

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

4.2.6 DI* Data acknowledge inverse

An acknowledge signal that is 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.

4.2.7 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.

4.3 Arbitrated message lines

In systems implementing an arbitrated message bus, these signals are used for bus master selection as described in the following subclauses.

4.3.1 AB[7..0]* Arbitrated message 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.

4.3.2 ABP* Arbitrated message bus parity

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

4.3.3 AP*, AQ*, AR* Arbitrated message synchronization

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

4.3.4 AC[1..0]* Arbitrated message condition

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

4.4 RE* Reset/Bus initialize

A line that is asserted by a module to initialize the bus interface logic of all modules to a predefined state, and to 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.

4.5 Central arbiter

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

4.5.1 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.

4.5.2 GR* Grant

This line is asserted by a central arbiter to grant bus mastership to a module.

4.5.3 RQ[1..0]* Request

One or both of these lines are asserted by modules to request bus mastership.

5. System/Bus initialization

5.1 Introduction

This section describes initialization procedures and functions that may take place during start-up of a Futurebus+ system. The intent is not to specify a particular start-up sequence, but to outline example Futurebus+ initialization procedures, to give guidance on the relative ordering of these tasks, and provide references to Futurebus+ and other related specifications. Specifically, clause 5.2 describes a typical system interface, discusses how system start-up is initiated and monitored by a system operator, and describes the relationship of these events to the start-up of the Futurebus+ backplane bus subsystem. Clause 5.3 details the steps needed to establish an initial configuration of Futurebus+ modules in preparation for the loading of system software. The emphasis in this section is on Futurebus+ specific initialization tasks. Standard, nonbus-specific initialization processes are provided in IEEE Std 1212-1991, CSR Architecture.

5.2 System initialization and operation summary

System start-up occurs after power up or full (hard) reset of the system and is often initiated, monitored, and controlled by an operator through the use of a set of switches, indicators, and/or an intelligent console. Once the system is operational, the system interface provides status information to operators and maintenance personnel that reflects the current health of the system. This information is a consolidation of status from the individual subsystems of the computer, including the Futurebus+ backplane bus subsystem. When a system shutdown is required for maintenance or other reasons, the same system interface supports the initiation and monitoring of a controlled shutdown of all resources, including the backplane subsystem.

5.2.1 Operator interfaces

No systemwide indicators, switches, or interfaces are specified for Futurebus+ systems analogous to those defined for Futurebus+ board front panels. Often, system indicators will be chosen by a designer to provide a corporate look. In this subclause, the emphasis is on describing the interface that an operator or maintenance personnel would find on a typical system and how this interface relates to what occurs on Futurebus+ during system start-up.

A system control console is often divided into two areas—an operator area and a maintenance area. The trend is for these to contain only a minimum number of indicators and switches. In the past, systems often had switches to enter data, and indicators to present the contents of critical registers. Systems designed more recently tend to use a standard workstation, personal computer, or a simple ASCII terminal for operator and maintenance interaction with the system. In the future, system control can be expected to be exercised from a remote location (sometimes referred to as *lights out operation*), which implies that a modem or a network interface will be necessary.

Regardless of the particular implementation of the operator interface, this recommended practice strongly encourages the use of ergonomic design, i.e., thoughtful provision of operating controls and indicators that are self-explanatory, clear, and unmistakable, even to nontechnical personnel who use and administer the system.

5.2.1.1 Operator access

The operator area will generally have a multiposition keyed switch or a set of switches and a small set of indicators. The switch is typically used to specify four operations—power off, power on, operate or boot, and enable remote operation. The indicators typically specify stand-by, run, and fault.

If remote operation capability is provided, a modem or other means for remote operation shall be supported. A modem connection also provides a mechanism for alerting remote maintenance personnel when a system problem exists, a port for downloading new software, and a means for collecting system usage statistics for remote evaluation. This remote access capability is, however, subject to installation site requirements for security; this topic is covered in more detail in section 14.

There are a number of ways the state of switches can be sensed and indicators set. Using a serial bus for this function is discussed in section 10.

5.2.1.2 Operator switches

For systems where the four switch operations listed above are deemed adequate, a single keyed switch can be used. The switch position itself does not generally initiate the action. The switch is read by an entity termed the *console manager process* that may reside outside the Futurebus+ subrack but that has access to the Futurebus+ RESET signal, **RE***. The console manager process may be capable of message-based communication with the operating system. It is assumed that a serial bus would provide the physical communication layer.

When the switch is placed in the power on position, the console manager process causes dc power to be applied to the system but a system (cold) start is not initiated.

System operation in this example begins when the switch is placed in the operate position. This is intended to cause a complete system reset before initiating operation. **RE*** must be asserted by the console manager process to cause a full reset in a manner specified by the Bus/System Management section in ISO/IEC 10857 : 1994. This causes each board to initiate its internal self-test. Boards that successfully pass self-test and that are monarch capable vie for the bus. The board that succeeds in acquiring the bus first becomes the configuration monarch. The competition for and duties of the configuration monarch are described in 5.3.4.

System shutdown can be initiated by either turning the keyed switch back to the power-off position or by use of a special shutdown switch. In either case a graceful system shutdown is expected. It is assumed that after system shutdown is initiated, the console manager process notifies the operating system which, in turn, notifies the individual nodes that a shutdown is to begin. This will normally require committing or journaling operations in progress, transferring modified cache lines to main memory, and transferring main memory to disk. Typically, as the nodes and associated I/O devices reach quiescence, they inform the system monitor. After all nodes have reached the quiescent state, the operating system either directly informs the power system monitor or it informs the console manager process at which time dc power can be removed from the nodes.

If it is to be possible to control the system remotely, then the keyed switch can have an additional position called remote. In this case the local console will usually be locked out and will only respond to commands that are received from the remote location. The indicators, which are also under the direct control of the console manager process, will not be illuminated. The information they present will be transmitted to the remote site.

While remote operation and remote maintenance can reduce the life cycle cost of a system, installations with high security requirements may not allow remote operation, since this can be viewed as an uncontrolled port into the system.

5.2.1.3 Operator indicators

The three indicators, referred to as run, stand-by, and fault, give a useful overview of the state of the system. The indicators can be used in combination. It is assumed that the indicators are directly controlled by the console manager process and that their status changes are based on the system data (such as switch settings and operating system messages) collected by the console manager process.

The run indicator is used to indicate that the system is operational. However, in a fault-tolerant system it is legitimate to have the fault indicator on and the run indicator on simultaneously, indicating that valid processing is taking place in the presence of faults. Having the fault indicator on indicates that a permanent fault has been detected. The system may still be able to function with this fault, in which case the run indicator would be on at the same time.

The stand-by indicator can be used to indicate that the system is in the process of starting up or shutting down. After power has come up, the console manager process could flash the stand-by indicator to indicate that the power supplies are operational and that processing (at least built-in-self-test) can begin. The console manager process sets the stand-by indicator on after the operate switch has been enabled. It remains on until the operating system informs the console manager process that normal system operation has begun.

When the operating system informs the console manager process that normal processing is underway, the console manager process will turn the stand-by indicator off and turn the run indicator on.

Depending on the characteristics of the system, an uncorrectable fault detected on any node will be reported to the operating system which, in turn, informs the console manager process. Based on this information and information the console manager process may have collected on its own from external modules, the fault light can be activated. If system operation is still proceeding, the run light will remain on. If remote maintenance is enabled, the console manager process will initiate the auto dial procedure. It will then inform the maintenance center of the fault and, depending on the sophistication of the diagnostics, what failed and when it needs to be replaced.

When a switch setting indicates that system shutdown is necessary, the console manager process will typically inform the operating system. It is also possible that the operating system will detect some event and notify the console manager process that the system is shutting down. The console manager process would then turn on the stand-by indicator. It would remain on until quiescence is reached. After the operations initiated by the system shutdown have been completed, the stand-by indicator can be flashed to indicate that dc power can be safely removed.

For systems that support live insertion (to avoid a system shutdown for maintenance), flashing the stand-by indicator can be used to indicate that the board to be replaced is quiescent. The board to be replaced should also present some visual indication of this fact.

5.2.1.4 Maintenance access

The switches and indicators associated with a maintenance area are often protected by a cover plate. For this discussion it is assumed the maintenance area consists of the keyed switch, a warm start switch, a maintenance connection, and an emergency off switch. It is again assumed that the console manager process is able to communicate with the operating system using the serial bus.

The keyed switch is used to activate this set of switches. The warm start switch causes the system to be reinitialized (system reset). Warm start differs from cold start (power-on reset) in that memory is not affected. Also, in a fault-tolerant system it is assumed that the operating system will be able to check error reports and reconfigure the system to disable faulty nodes. Warm start is often used to recover from software faults that caused the system to go into an indeterminate state. Warm start is indicated to Futurebus+ boards by signaling "Bus Initialize" on the reset line in the manner specified in the Bus/System Management section in ISO/IEC 10857 : 1994. The necessary circuitry is assumed to be part of the console manager process and can be activated without cooperation from the operating system.

The maintenance connection permits maintenance personnel to attach a maintenance device to the console manager process. This may be a terminal or personal computer. Alternatively, the maintenance device may be a special device with a processor that connects directly to the serial bus and enables boundary scan functions on individual chips. In systems where cost is a critical factor, the maintenance function should be reachable from a user's port.

When activated, the emergency switch causes dc power to immediately drop. No attempt would be made to save the state of the system. Electrical codes sometimes require an emergency off switch to be located away from the equipment.

5.3 Bus initialization

Initialization of the Futurebus+ during a system start-up as described above begins with assertion of the Futurebus+ **RE*** signal by the console manager process. On reset, all Futurebus+ nodes (there may be up to two nodes per Futurebus+ module) enter built-in-self-test, which brings all boards to an initial known state. After power up, built-in-self-test, and upon the release of **RE***, a single configuration process or configuration monarch is appointed via a selection algorithm or by convention to oversee further initialization of the system. Specifically, the configuration monarch may be responsible for coordinating further diagnostic tests at the node and system levels, establishing an initial Futurebus+ operational configuration, and configuring physical memory. The configuration monarch will then boot the operating system software and transfer control. In this clause, these major components of Futurebus+ bus initialization are described. The tasks described and the relative order in which they are presented is intended to represent a likely (but not required) set of key initialization functions and order of execution.

5.3.1 System reset

System start-up begins after the occurrence of a 100–200 ms pulse on the Futurebus+ **RE*** line as specified in the Bus/System Management section in ISO/IEC 10857 : 1994. This event causes a full reset of the logic on all Futurebus+ nodes.

5.3.2 Module identification

System reset, in turn, initiates built-in-self-test diagnostics on all Futurebus+ nodes. Depending upon the module's design, a node may need to first identify itself and learn its capabilities in order to invoke proper diagnostic routines. In this case, the node must access its bus-specific ROM entries located in CSR ROM. Futurebus+ defines a standard nameplate that contains a hierarchical description of the module, the module's node(s), the node's unit(s). The description of each of these entities includes a manufacturer's identifier, hardware and software revision numbers, and other configuration information as specified by the Node Management section in IEEE Std 896.2-1991.

5.3.3 Built-in-self-test

Based upon information in the CSR ROM, proper diagnostic programs are executed by the on-board processing resource. Proper test result codes are used to check the signatures of self-testing logic.

These power-up diagnostics are of the INITIALIZE category as defined by the node's TEST_START CSR. Results of a node's initialization diagnostics are recorded in its TEST_STATUS CSR.

For details on the format and operation of these diagnostic test CSRs and CSR ROM entries, consult the IEEE Std 896.2-1991, IEEE Std 1212-1991, and section 9. of this recommended practice.

5.3.4 Configuration monarch selection

To establish an initial bus configuration after a cold start, a single configuration processor or configuration monarch must be appointed to run the configuration-initialization process. In some systems, the selection of the configuration monarch processor is made by the hardware or the operator. In others, a selection algorithm is needed to

unambiguously chose one processor as the configuration monarch. The Futurebus+ arbitration message facility allows the configuration monarch processor to be dynamically selected. This selection normally occurs immediately after power-up BIST.

In a warm start, the previous configuration monarch can re-assume the role, or the role of configuration monarch can be recomputed using the arbitration message facility. The identity of the processor to run the configuration monarch process may be changed during system operation, or multiple processors may execute the configuration-management process after initialization.

5.3.4.1 Automatic selection algorithm

With the exception of Profile F (described in IEEE Std 896.2-1991), Futurebus does not specify a particular configuration monarch selection algorithm. This subclause describes an example configuration monarch selection algorithm useful for single bus segment Futurebus+ implementations. This algorithm employs the arbitrated message capability of Futurebus+ to support a configuration monarch competition among multiple configuration monarch contenders immediately after a system reset. This algorithm does not select the specific processor to run the configuration monarch process. Rather, from the set of candidate modules, the algorithm unambiguously selects a module. It is the responsibility of the vendor to design the module so that, once the module is selected, the configuration monarch function is assigned to a specific node and processor on the module. It is noted that a more elaborate algorithm is required to guarantee that only a single configuration monarch is chosen in multiple bus segment (bridged) architectures.

When a system is first powered on, its modules perform a self-test and hold the reset line **RE*** asserted until they are ready to participate in bus activity. When the reset signal is released, a node that is configuration monarch capable may try to become the configuration monarch. Nodes establish whether or not they are configuration monarch capable by checking a vendor-defined entry in ROM.

Only one configuration monarch should be selected as a result of the configuration monarch competition and the fact of its selection should be conveyed to all other potential candidates. The selection and broadcast of the results should be a single indivisible operation to prevent two modules from simultaneously assuming they are to execute the configuration monarch process. This is achieved by having every configuration monarch capable module attempt to send an arbitration message saying it is the configuration monarch. The winner of the arbitrated message competition sends the arbitration message and becomes the new configuration monarch.

Table 2— Configuration monarch competition message

Bit	cn7	cn6	cn5	cn4	cn3	cn2	cn1	cn0
Pass 1	1	1	1	1	1	1	1	1
Pass 2	1	1	0	GA4	GA3	GA2	GA1	GA0

The specific arbitration message sent during the arbitrated message competition is illustrated in table 2. GA[4...0] is the geographic address of the module. It is reiterated that configuration monarch competition is performed between modules. If a module contains two nodes, or if a node contains multiple processors, it is the responsibility of the module designer to assure that one and only one processor is initialized as configuration monarch. By convention, the winner of the competition is the highest arbitration number.

The configuration monarch competition will be initiated by the first configuration monarch capable module that completes its self-test after reset has been released. (Reset must be released within 1 s or it is assumed stuck asserted). Even though the reset line has been released, a module may not be ready to initiate transactions. However, by releasing **re***, a configuration monarch candidate indicates that it can record the results of a configuration monarch arbitration competition even if it is not ready to participate.

The results of configuration monarch competition may be stored in a vendor-defined CSR register indicating whether the module was the configuration monarch winner or configuration monarch loser. Before attempting to become configuration monarch, a node checks its configuration monarch winner/loser status to determine if a configuration monarch has already been chosen. If configuration monarch loser status is indicated, then the processor backs off and waits to be configured by the existing configuration monarch. Otherwise, the processor proceeds with the configuration monarch competition. On winning a configuration monarch competition, a processor records its configuration monarch-winner status and proceeds with configuration monarch duties. On losing the competition, a processor records its configuration monarch loser status and waits to be configured.

It should be noted that monarch capable modules that are also live insertable shall have a mechanism for guaranteeing that they boot in a manner that sets the configuration monarch-loser CSR bit when live inserted into a functioning system.

5.3.5 Configuration monarch duties

Once a Futurebus+ module becomes configuration monarch it shall perform a number of initialization tasks to configure all nodes on the local bus prior to the loading of system software. The extent of these configuration monarch duties is profile and system dependent. This subclause describes several configuration monarch duties that will be useful to many profiles and applications.

5.3.5.1 Node population

The monarch's first task is to establish the system's physical module/node population. To accomplish this the configuration monarch simply attempts a quadlet read-unlocked transaction to a core Futurebus+ CSR (e.g., NODE_ID CSR) at each of the 62 possible node addresses on the local Futurebus+. Valid responses to the read signify the presence of a node. No response or an invalid response signify that a node is not present or that it is nonfunctional.

Table 3 lists an address sequence that a configuration monarch could use to accomplish the above described task. The target CSR for each of the 62 possible nodes is NODE_ID. The local bus address of 1023 is used in each address.

Table 3— Typical polling sequence to establish node population

Slot number	Futurebus+ NODE_ID CSR Physical address
1	0xFFFC0008, 0xFFFC1008 (Nodes 0,1)
2	0xFFFC2008, 0xFFFC3008 (Nodes 2,3)
3	0xFFFC4008, 0xFFFC5008 (Nodes 4,5)
	
	
15	0xFFFD0008, 0xFFFD0008 (Nodes 28,29)
	
	
31	0xFFFFC008, 0xFFFFD008 (Nodes 60,61)

The configuration monarch records the success/failure of each quadlet read to compile a list of active nodes, thereby establishing the initial backplane physical population.

5.3.5.2 System built-in-self-test

Once the physical configuration is known, the configuration monarch may poll each (active) node to determine its built-in-self-test status. To do this the configuration monarch reads the TEST_STATUS CSR. The configuration monarch records which nodes passed their INITIALIZE diagnostics as indicated by the CAT field in this register. Nodes with TEST_STATE equal to ACTIVE when polled by the configuration monarch are reread at regular intervals for 10 s (as specified in IEEE Std 896.2-1991) to determine their final diagnostic status (either passed, failed, or time-out). The configuration monarch thus establishes a list of good Futurebus+ nodes that it must configure. (Nodes may undergo further extended diagnostic tests, system diagnostic tests, and/or manual diagnostic tests under command of the configuration monarch in order to establish their ultimate operational health.)

5.3.5.3 Arbitration mode

In systems with nodes that support both central and distributed arbitration, the configuration monarch is responsible for configuring all nodes to arbitrate for Futurebus+ using the same mode. This is done by reading the NODE_CAPABILITIES_EXT CSR of all nodes, making a decision as to which mode to invoke, and then enabling this mode by setting or clearing the CENTRAL_ARBITER bit in each node's LOGICAL_COMMON_CONTROL CSR. Arbitration compatibility errors are reported by the monarch through the operator interfaces.

5.3.5.4 Arbitration bus parameters

At system initialization, the ARBITRATION_DELAY_REGISTER CSRs are set to default values. Before nodes can be optimized for distributed arbitration or arbitrated messages, the configuration monarch must program the ARBITRATION_DELAY_REGISTER CSRs. The configuration monarch must program these registers for each master-capable node (including itself).

The configuration monarch programs the ARBITRATION_DELAY_REGISTER by performing the following procedure:

- a) For each good node in the system, read the MODULE_LOGICAL_CAPABILITY CSR and determine if the MASTER_CAPABLE bit is set. If set, then read and record the node's COMPETITION_INTERNAL_DELAY_REGISTER CSR.
- b) Determine the maximum COMPETITION_INTERNAL_DELAY value in the set of recorded COMPETITION_INTERNAL_DELAY_REGISTER CSRs.
- c) Obtain the value of the bus propagation delay from local ROM and calculate the worst-case competition settling time for a node during distributed arbitration.
- d) Write the COMPETITION_SETTLING_TIME field of the ARBITRATION_DELAY_REGISTER CSR of each master-capable node with the computed worst-case value.

5.3.5.5 Program arbitration priority

Futurebus+ supports priority-based arbitration in both central and distributed mechanisms. Nodes employing this arbitration mechanism may need to have their priority established at system start-up. To perform this function, the configuration monarch programs the node's 8-bit priority field in the NODE_ID CSR. The node may use this priority number in arbitrated message competitions.

5.3.5.6 Program propagation delay

In all Futurebus+ nodes, regardless of the arbitration mechanism used, the configuration monarch programs the BUS_PROPAGATION_DELAY_REGISTER CSR. This CSR is written with the value the node is to use for its wire-OR integration delay time (two times the bus propagation delay time). A default value of 0×20 (7.45 ns) is used by the node if the configuration monarch does not write this register.

5.3.5.7 Parallel protocol configuration

Next, the configuration monarch seeks to establish a Futurebus+ parallel protocol configuration that: a) supports basic communication among all nodes in the system and b) supports optimized communication among modules compliant to a particular profile.

5.3.5.7.1 Basic parallel protocol

The basic parallel protocol-related CSRs that the configuration monarch must check or configure to make a system work are listed in table 4. The intent here is to provide the designer a reference for all the capability, control, and status registers and the pertinent bits/fields associated with each parallel protocol function. The use of these CSRs and ROM entries to initialize the parallel protocol is detailed in the Bus/System Management section in ISO/IEC 10857 : 1994 and the Node Management section in IEEE Std 896.2-1991.

Table 4— Basic parallel protocol

Register name	Bit/Field
Data length NODE_CAPABILITIES_EXT LOGICAL_MODULE_CONTROL	DATA_LENGTH_n COMPELLED_DATA_LENGTH
Data width MODULE_LOGICAL_CAPABILITY LOGICAL_MODULE_CONTROL	NBIT_DATA_CAPABLE DATA_WIDTH
Address NODE_CAPABILITIES MODULE_LOGICAL_CAPABILITY LOGICAL_MODULE_CONTROL NODE_IDS	EXT_ADR_CAP UNIT_ADR_CAP INT_ADR_CAP 64-BIT_ADDRESS_CAPABLE 64-BIT_ADDRESS BUS_ADDRESS
Time-out TRANSACTION_TIMEOUT	
NOTE — Setting of bits in the LOGICAL_MODULE_CONTROL CSR requires a protection of the register so as not to disrupt the state of other bits that may be set in the register.	

5.3.5.7.2 Parallel protocol options

There are many additional functions related to the parallel protocol that the configuration monarch may need to configure to support higher-level protocols or optional features on a module. The CSRs (and applicable bits/fields) associated with these functions are summarized in table 5.

Table 5— Optional parallel protocol

Register name	Bit/Field
Split transactions MODULE_LOGICAL_CAPABILITY LOGICAL_COMMON_CONTROL SPLIT_TIMEOUT	SPLIT_ACCEPTOR SPLIT_INITIATOR SPLIT_ENABLE
Packet mode MODULE_LOGICAL_CAPABILITY LOGICAL_COMMON_CONTROL	PACKET_LENGTH_N MULTIPLE_PACKET_CAPABLE PACKET_CAPABLE PACKET_LENGTH_N_ENABLE MULTIPLE_PACKET_MODE HIGH_PACKET_SPEED LOW_PACKET_SPEED PACKET_ENABLE HIGH_SPEED_ENABLE
Tag MODULE_LOGICAL_CAPABILITY LOGICAL_MODULE_CONTROL	NUMBER OF TAG LINES NUMBER OF TG[]* LINES ACTIVE TAG_ENABLE
Locked transactions MODULE_LOGICAL_CAPABILITY	FETCH_ADD_CAPABLE COMPARE_SWAP_CAPABLE SWAP_CAPABLE
Retry BUSY_RETRY_COUNTER_CAPABILITY BUSY_RETRY_COUNTER BUSY_RETRY_DELAY_CAPABILITY BUSY_RETRY_DELAY ERROR_RETRY_COUNTER_CAPABILITY ERROR_RETRY_COUNTER ERROR_RETRY_COUNTER_CAPABILITY ERROR_RETRY_DELAY	
NOTE — Setting of bits in the LOGICAL_MODULE_CONTROL and LOGICAL_COMMON_CONTROL CSR requires protection of the register so as not to disrupt the state of other bits that may be set in the register.	

5.3.5.8 Higher layer protocol configuration

Next, the configuration monarch may program nodes to support Futurebus+ protocols that are layered on top of the parallel protocol. The CSRs (and applicable bits/fields) associated with these functions are summarized in table 6.

Table 6— Layered protocols

Register name	Bit/Field
Cache transactions MODULE_LOGICAL_CAPABILITY LOGICAL_MODULE_CONTROL	CACHE_CAPABLE CACHE_ENABLE
Message passing MODULE_LOGICAL_CAPABILITY MESSAGE_FRAME_SIZE MESSAGE_PASSING_SELECT_MASK	MESSAGE_CAPABLE
NOTE — Setting of bits in the LOGICAL_MODULE_CONTROL and LOGICAL_COMMON_CONTROL CSR requires protection of the register so as not to disrupt the state of other bits that may be set in the register.	

5.3.5.9 Configure Futurebus+ physical memory space

The configuration monarch will normally configure Futurebus+ accessible memory on all good nodes. The task is to build a memory table, assigning shared memory space to each Futurebus+ node that requires it. Memory allocation to nodes may proceed in any order.

Each node's Futurebus+ memory requirement is visible to the configuration monarch via the CSRs for memory unit address space. If the ROM entry is nonzero, the configuration monarch configures the node into the Futurebus+ memory space by setting the MEMORY_BASE and MEMORY_BOUNDS CSRs as specified in the Node Management section in IEEE Std 896.2-1991. Memory is normally allocated in contiguous blocks beginning at physical address 0x00000000. The configuration monarch is responsible for ensuring that each node's memory allocation begins on a 4 kb address boundary. The configuration monarch must also ensure that no memory is allocated at address locations reserved for private space and register (CSR) space.

The CSRs for extended units address space indicate that the node requires (additional) Futurebus+ address space other than for memory. The configuration monarch allocates space by setting the UNIT_BASE and UNIT_BOUNDS CSR per the Node Management section in IEEE Std 896.2-1991.

The INDIRECT_ADDRESS and INDIRECT_DATA CSRs do not require initialization by the configuration monarch process. However, these registers shall be protected (e.g., memory-resident locks) if concurrent accesses by multiprocessors are allowed.

5.3.5.10 Memory initialization/enable

At this point, address space is allocated but not enabled. The configuration monarch is responsible for enabling a node's memory and unit space by setting the enable (ENB) bit in the MEMORY_BASE CSR. Only after this bit is set may the node respond to memory transactions (other than CSR transactions) over the bus.

The configuration monarch may be required to clear system memory by writing all zeroes to every address in the allocated memory space. This initialization of memory is an optional configuration monarch task, since it may be performed as a part of a node's built-in-self-test or may not be required at all.

5.3.5.11 Extended diagnostics

Once the system's memory has been configured, the configuration monarch may be required to coordinate the execution of node EXTENDED tests. These tests check circuitry/firmware not covered by initialization diagnostics. Extended tests are distinguished from initialization diagnostics in that they may make use of Futurebus+ and other module I/O paths or take an extended period of time to execute.

The master and arbitration functions of the node under test were previously enabled via setting the MASTER_ENABLE and ARBITRATION_ENABLE bits of the LOGICAL_MODULE_CONTROL and LOGICAL_COMMON_CONTROL CSRs respectively. Configuration monarch coordination of extended tests at a minimum involves the following:

- a) Write the node's TEST_ARGUMENT CSR with the target memory address for the extended tests.
- b) Write the node's TEST_START CSR with the category field programmed for extended tests.
- c) Monitor the TEST_STATUS CSR of the node under test.

In addition to EXTENDED tests, the configuration monarch may be responsible for invoking and supervising the execution of SYSTEM and MANUAL diagnostic tests. A detailed guide on the use of the test registers (specified in IEEE Std 896.2-1991 and IEEE Std 1212-1991) is found in section 9. of this recommended practice.

5.3.6 Operating system boot

Upon completion of its bus programming and other system configuration duties, the configuration monarch will typically search in ROM to find the code source device, perform an initial program load, and boot the operating system (refer to IEEE Std 1275-1994). This terminates the monarch's responsibilities until a system reset, live insertion, or failure requires that a reconfiguration of the system take place.

6. Byte lane mapping

6.1 Introduction

Futurebus+ is designed to support both big endian and little endian architectures. Both types can be implemented with the same cost and performance. Systems can run with no run-time overhead if they use the same endian. If the systems use mixed endians, they can run with byte swapping required. Initialization and configuration may require byte swapping.

Futurebus+ supports big and little endians by fixing the byte lane address map, such that all modules agree where to find any byte offset on the bus. This address invariance approach implies that all processors see the same data structures or components of data structures at the same addresses. However, not all processors necessarily see primitive data items that are not encoded in their own native format.

The remainder of this section gives some history behind the endian decisions, and shows some possible implementation details that might not be obvious.

6.2 Endian history

Computer designers assign names and numbers to all objects that they use. For example, every bit on an address or data bus is labeled with a number. Byte-addressable architectures address every byte with a number.

However, labels are arbitrary. There is no principle of nature favoring left-to-right, right-to-left, or even top-to-bottom labeling orders. As a result many different self-consistent labeling systems have been invented.

Computer architectures are now divided into two camps, usually called *big endian* and *little endian*. D. Cohen (see [B1])⁸ derived the names from Jonathan Swift's book *Gulliver's Travels*. In that satire, thousands died in a religious war about whether to break eggs from the big end or the little end.

⁸The numbers in brackets correspond to those of the bibliographical references in annex B.

Architecture descriptions in English-language publications display data structures with the most significant bit on the left, and the least significant bit on the right. Big endian architectures label bits and bytes from left-to-right. Little endian architectures label bits and bytes from right-to-left. This seemingly innocuous difference has cost hundreds of man years of design complications and millions of dollars in losses from bug-filled products.

Newcomers to this issue tend to have a common progression. They start by disbelieving that anyone would do anything different than the first system that they were exposed to. There is a great deal of arguing and miscommunication. Strong arguments arise about the best or most natural endian. First they have a desire to force everyone else to convert to their favorite endian. Next they try to isolate their systems from alien influences. They then try technical tricks to make the problem magically disappear. If the problem does not disappear, a data invariant mapping is tried. If that does not work, then an address invariant mapping is tried. If the problem still exists, then compiler support is desired. Most people who have to deliver systems that cross endian boundaries end up believing that the problem will never disappear, will always be a source of hassles and bugs, but is not a big problem in practice.

There is no way to technically arrange matters so that the instruction set architecture's endian decision is not visible to C or assembly language programmers. Designers can make text strings readable in either endian, or they can make binary integers readable by either endian, but not both. (If they could, the two endians would be the same!) Note that the problem is in the internal representation of data. There are many other issues surrounding compatible representations. For example, there are incompatible memory alignment rules (e.g., floating point word aligned on a byte versus quadlet boundary) on different machines, which are pragmatically as important sources of portability problems as endian problems. There are also problems with ASCII versus EBCDIC, ones complement versus twos complement, floating point formats, bit field packing orders, and so on.

There is no way to make all machines the same endian. There is too much commercial momentum behind the two. The two largest computer architectures (in terms of dollars) are the IBM® mainframes (big endian) and the IBM personal computers (little endian). Each of these architectures have hundreds of billions of dollars of endian-sensitive software that will not be soon replaced. The endian problem will probably be present for many years to come.

6.2.1 Endian-sensitive standards

Since each labeling system is internally consistent, why is there a problem? The problem occurs when attempting to make vendor-independent interfaces.

Isolated computer systems are becoming a thing of the past. Networking can connect most computers over phone lines or local area networks. Bus-based systems connect boards of different styles into one system to cooperatively solve problems for one system integrator. Thus, the design community is challenged to produce computer systems that effectively cooperate over endian dialect barriers.

Every industry standard interface point must be well defined, so that multiple teams of implementors can make interoperable products. Many aspects of the data items in the interface must be specified, including field size, field alignment, and any special coding. Every multi-byte data field must have its endian interpretation specified.

Standards organizations such as the IEEE are obligated to make standards that do not unduly benefit one commercial camp at the expense of another. The open voting rules of the standards process are unlikely to allow arbitrarily one-sided standards to be adopted.

Whether a processor's native data structures are big endian or little endian is beyond the scope of the Futurebus+ standards. The native data structures might not even be visible at the bus interface.

6.2.2 Data invariance

Designers have long faced multiple endian designs. Two ways have been used to map together opposite endian designs.

The first way is to map words together. For example, assume a 32-bit bus. Map corresponding 32-bit words together, with the most and least significant bits lining up. In this way, 8-bit, 16-bit, and 32-bit integers are mapped so that either endian processor can use them without byte swap overhead. This is called *data invariance*, and has been used many times in multiple endian designs.

Hardware designers tend to like data invariance, because the hardware implements operations on primitive data structures. However, almost all implementation effort is in software. Software spends little time in dealing with single arithmetic operations on single primitives. Instead, it spends most of its time dealing with complex data architectures built out of arrays and structures.

Unfortunately, data invariance scrambles data structures and arrays. Data structures are scrambled, requiring multiple definition files or conditional definitions to attain portability. Arrays are scrambled, with consecutive indexes selecting scattered items.

In particular, human-readable text is shuffled, with the shuffling depending on the starting offset of the string. In fact, a single contiguous string on one machine can be seen as multiple disconnected strings on the other one.

Also, the scramble depends on the word width used to map the words together. Some implementations map data words together based on the bus width. In this instance the software needs to know the width of the bus. Moreover, it may need to know the width of all intermediate buses in a system with bridges.

6.2.3 Address invariance

The other approach is to preserve byte addresses when mapping between Endian domains. This is called *address invariance*, and has been used many times in multiple endian designs. In this situation all data structures and arrays keep their original order, all human-readable text is immediately usable, and buffers defined by starting address and length are interpreted the same way by all parties.

This principle is usually extended across all communication media. It does not matter whether the media is a bus, a bridge between buses, a LAN, a modem, a magnetic tape, or any other means. The interpretation of the data is independent of the transportation details.

Address invariance does not magically solve the endian problem. Multi-byte binary integers are not immediately usable by the processor. (If both text and binary integers were simultaneously in the native format of two different machines, they would be the same endian!) Address invariance makes human-readable text portable, and gives a constant, stable basis to solve the more involved data encoding portability problems.

Therefore, each software interface designer must define the endian interpretation of each multi-byte data field. Software for a processor that cannot directly interpret that must put in a byte swap macro.

Ultimately, this problem should be automatically handled by compilers. The type declaration would specify the encoding to be used in exported or imported data. The compiler would automatically generate the necessary cross addressing or byte swap code. However, this solution is not supported in current compilers.

6.2.4 Hybrid endian

Many systems are not purely one endian or another. *Hybrid endian* architectures have little endian bit labels on schematics, while all programmer-visible manipulations of data within the architecture are big endian.

Hybrid endian architectures are popular in many microprocessor architectures, including the Motorola MC68000 and SPARC chips, and in the VMEbus and SBus standards.

6.2.5 Cross endian

A response to the endian issue is to support both endians through module design time configuration. This allows manufacturers to increase the market for one design, by allowing either camp to use the module.

A more radical approach is to support both endian camps dynamically. For example, a processor could have an endian interpretation bit in its processor status word. On every context switch the processor could chose the appropriate interpretation. The system may well be a native big endian system that has little endian interpreters for other environments.

Products that can be configured, either statically or dynamically, to support either big endian or little endian interpretation are called *cross endian*.

Cross endian products may use either address invariance or data invariance. Data invariance tends to be useful only at design time. Address invariance is also useful in dynamically accommodating interface or emulator endian requirements.

6.3 Addressan

There is still another endian variation. When addresses and data buses are multiplexed onto common pins, there are several alternatives. The least significant byte (Byte 0) of data can be multiplexed with either the most significant or the least significant part of the address. A bus that multiplexes the most significant (big) end of addresses with the lowest address data byte is called *big addressan*. A bus that multiplexes the least significant (little) end of addresses with the lowest address data byte is called *little addressan*.

Big endian designers prefer big addressan buses because the documentation is simplified. More importantly, integers and addresses have the same encoding on the bus. Similarly, little endian designers prefer little addressan buses.

Addressan and endian are independent. It is quite possible to make any combination of big endian and little endian architectures on big addressan and little addressan architectures. For example, Apple's Macintosh® uses the big endian MC68000 architecture on the little addressan NuBus. According to designers from Apple, this costs nothing. All that was required was to connect the traces to the bus pins appropriately.

Since Futurebus+ has a multiplexed address and data bus, it has some addressan characteristics.

6.3.1 Futurebus-1987

Futurebus-1987 was an early version of Futurebus+. It had only one width for both addresses and data. It had an addressing convention that avoided putting integer labels on bytes within words. This allowed Futurebus-1987 to be neutral to the big-endian-little-endian debate. A big endian advocate could implement a big addressan variant, and a little endian advocate could implement a little addressan variant. Any interoperability was left up to the system implementor.

6.3.2 Futurebus+ evolution

Futurebus+ added multiple address widths and data widths. Suddenly it was no longer possible to stay agnostic. A memory module could be asked to store “AB” in one 64-bit word, and then could be asked to return it as a sequence of 32-bit words. The issue then arose as to whether it should return “A” and then “B”, or “B” and then “A”. This is because whenever a standard supports multiple width access to the same data, it has to specify whether left or right is first or second; that is, it has to be addressan sensitive.

Therefore, Futurebus+ had to make a choice that Futurebus-1987 was able to avoid. The initial stance was to remain silent, and allow implementors to do anything that they desired. This led to major concerns about interoperability. After lengthy discussion, address invariance was chosen over data invariance.

The next stance was to have two options. Futurebus+ could be either big addressan or little addressan as the system implementor desired. All modules would record their addressan capabilities in ROM. All modules would implement at least one addressan, but could implement both. The standard was optimized for low-cost homogeneous endian systems.

As time went on, module manufacturers wanted to make one module to satisfy both endian camps. (No sense in halving potential market size and reducing profits.) Both Profile A and Profile B settled on requiring both Addressan variants to be built. However, this was expensive, requiring data path multiplexers and complex initialization software. The participants came to realize that having any one addressan was far superior to supporting both. Therefore, Futurebus+ changed to mandate one fixed addressan.

6.4 Futurebus+ addressan standard

All labeling on Futurebus+ is consistently little endian. Bits and bytes are labeled from right to left. Byte 0 is composed of bits 7:0, with bit 7 the most-significant bit and bit 0 the least-significant bit. Bit 0 is the least-significant end of the address field.

Note carefully the use of the word *labeling* in the previous paragraph. Applications are allowed to use absolutely arbitrary encodings in data fields. Big endian architectures can be put on Futurebus+ with no cost or performance penalties (except for a minor cost penalty in some implementations of big endian architectures with internal multiplexed address and data buses).

Technically the bit significance within bytes is not specified. However, all architectures are expected by convention to have the least-significant bit within a byte have the lowest bit label on the backplane.

Technical arguments were invented to show that one or the other addressan was more natural. However, the choice of little addressan over big addressan was arbitrary. There is no damage to either the big endian or little endian commercial camps, so there is little incentive to debate forever.

6.4.1 CSR registers

Futurebus+ uses IEEE Std 1212-1991 for nonbus-specific CSRs. It has multiple byte integer fields, and quite properly has specified their endian interpretation. IEEE Std 1212-1991 is consistently big endian.

Many of these core CSR registers will only be used during bootstrap. In terms of code, that is not performance critical. The byte swap macros that are probably required in little endian modules should not impact the operational performance of the system.

6.4.2 Unit architecture

Run time I/O performance could be impacted by formatting I/O control parameters in an alien format. The format of these parameters is specified in CSR unit architectures.

The Futurebus+ profiles recommend that unit architecture implementors have two variants of their control structures. One variant would be used by big endian system integrators, and the other would be used by little endian system integrators.

These variants are only needed in control parameters, not in the actual data stream. Address invariance causes buffers to have the same interpretation as a byte stream in all architectures.

6.4.3 Message passing

The lowest level message-passing service on Futurebus+ delivers a packet of up to 64 bytes. The bytes in the packet are delivered in the same order that they were sent, which is another way of saying that they are address invariant. Each protocol layer shall define the endian interpretation of any multiple byte data field that is used in messages. The message level of the protocol defined in ISO/IEC 10857 : 1994 uses little endian fields.

6.5 Relabeling

One way to correctly implement Futurebus+ addressing conventions is to relabel signals at some convenient point in the module. The bit significance of addresses and data will differ in big endian designs, so the relabeling is shown on both the local data bus and the local address bus.

The wiring pattern is easiest to show in diagrams. The Futurebus+ bits are displayed in a vertical column (to avoid showing left or right preferences). The local address and data bus are displayed in a horizontal row, with the most significant bit on the left and the least significant bit on the right. The numbers in the boxes are the byte lane offsets. The letters in the boxes are the address offsets, with “l” for the little end of the address, “b” for the big end of a 32-bit address, and “B” for the big end of a 64-bit address. The small numbers alongside the boxes show the order of the bit labels. The wires for just the most and least significant bits of each byte are shown, to avoid further clutter.

6.5.1 Little endian relabeling

Since Futurebus+ uses consistent little endian labeling, no relabeling is required. Figure 3 shows the wiring pattern.

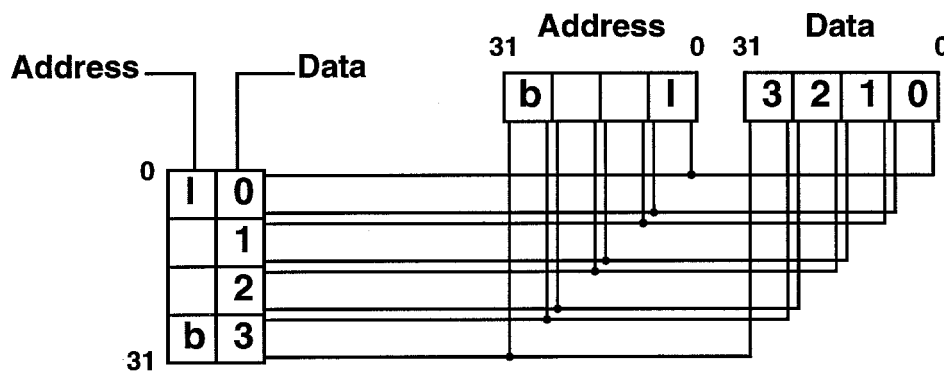


Figure 3— Little endian wiring

6.5.2 Big endian relabeling

The overall pattern of figure 4 is the same as figure 3, except that local labeling is done from left to right. The wiring of the address path is physically the same as in the little endian case, but the labels are different. Futurebus+ AD(0), the least-significant bit of the address, becomes ADDRESS(31). The wiring of the data path is substantially different. Futurebus+ AD(0), the least-significant bit of byte 0, becomes DATA(7) the least-significant bit of byte 0

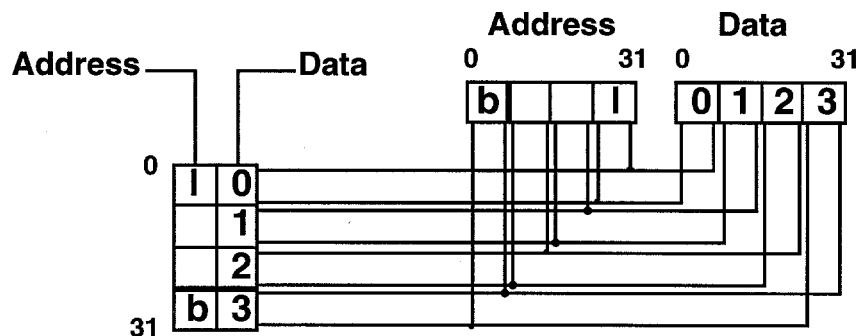


Figure 4— Big endian wiring

The wiring pattern of figure 5 is physically identical to that in figure 4. However, in hybrid endian architectures the local bit labeling is done from right to left. Futurebus+ AD(0), the least-significant bit of byte 0, becomes DATA(24), the least-significant bit of byte 0. Futurebus+ AD(0), the least-significant bit of the address, keeps the same bit label as ADDRESS(0).

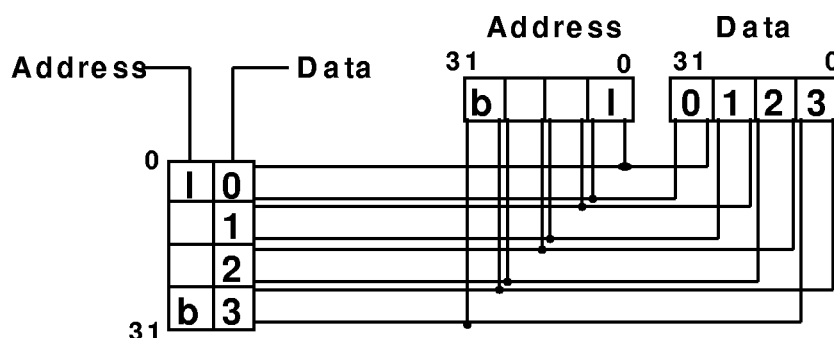


Figure 5— Hybrid endian wiring

6.6 Multiplexers

If the local bus on a Futurebus+ board has separate address and data buses, then the addressan chosen for the backplane has no part count or gate impact. The connections of the local address bus, local data bus, and Futurebus+ interface bus are just traces.

If the local bus has multiplexed address and data buses, and the multiplexing order is different from that chosen for Futurebus+, then active multiplexing logic must be used to realign the buses. Futurebus+ is little addressan, therefore, a board with little addressan multiplexing does not need any additional logic. A big addressan multiplexed board does require additional gates in its interface device so as to multiplex the addresses the board designer desires.

This penalty to big endian designers who also use multiplexed address and data buses was judged to be less than the penalty of active multiplexers on every board.

6.6.1 Big addressan multiplexers

As a worst-case example, consider a big endian board with multiplexed 64-bit address and 64-bit data. The board must have multiplexers to byte swap the address and data. The board must also have multiplexers to swap the 32-bit data field that must be supported on all Futurebus+ designs.

The wiring diagram just shows byte lanes. The bit treatment follows the pattern already shown. The small rectangles represent bidirectional buffers. There are many arrangements that implement the same function. The design rule chosen here is that there is at most one buffer between the Futurebus+ interface chips and the local multiplexed address and data bus.

The top block of four byte buffers handles the 32-bit byte swap required in all 64-bit designs. The next block of 8 byte buffers handles the address path. The last block of eight byte buffers handles the 64-bit data path and the aligned 32-bit data path. Using 18-bit wide bidirectional buffer parts yields 10 buffer parts. The equivalent little endian design would share the 64-bit address and data paths, yielding 6 buffer parts.

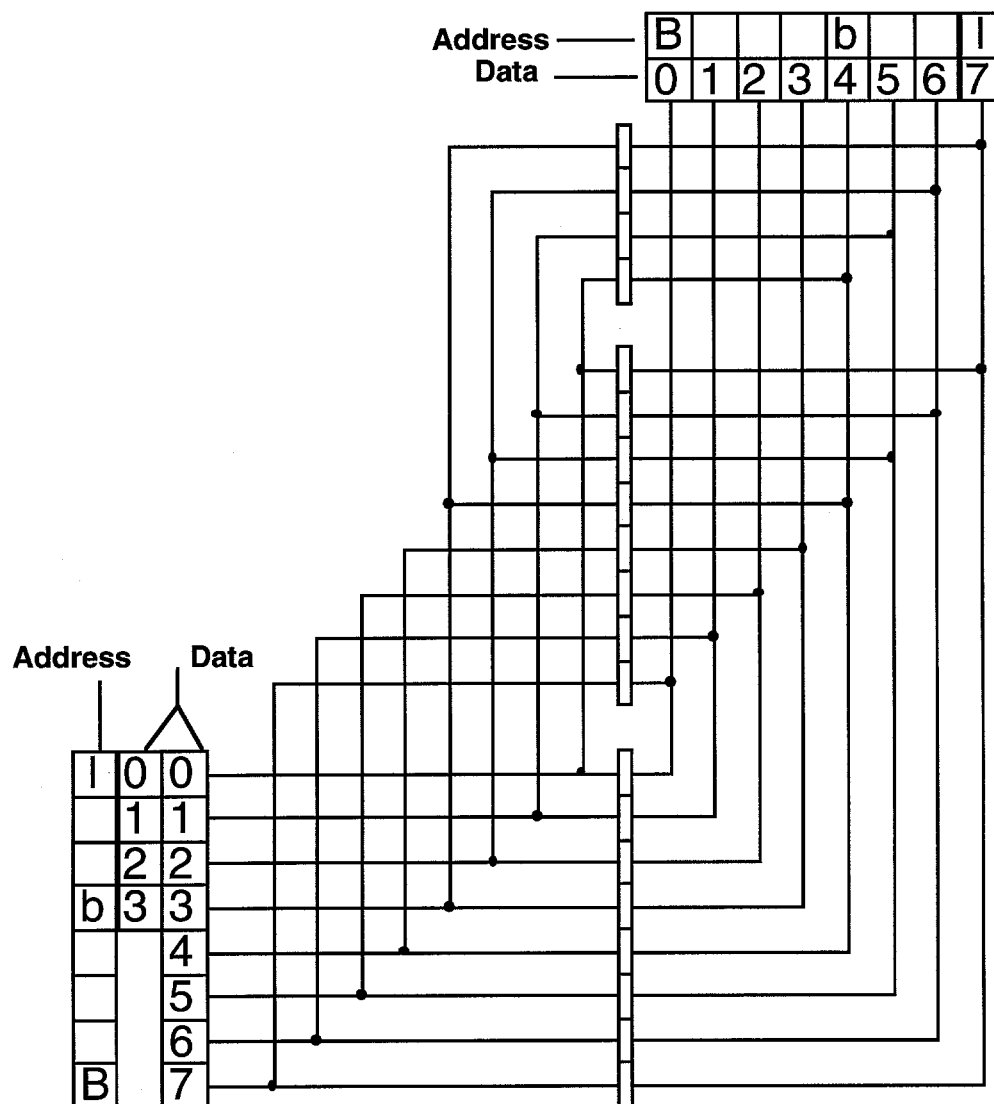


Figure 6— Big addressan 64-bit multiplexing

A more likely implementation would have the logic inside the data path ASICs on the board. In this case, the data path ASIC would have a large set of multiplexers or shift barrels to handle the multiplexing function. The data path may well need to be split into multiple parts to handle the pin limitations of some packaging techniques. The stride should be chosen such that the multiplexing connections are all inside the ASIC. That is, instead of handling bits {0,1,2,3,4,5,6,7} in one part, handle bits {0,8,16,24,32,40,48,56}.

If the board supports DMA between unaligned buffers, a more complex shift barrel is needed to align the data source and destination of the transfer. Adding the addressan multiplexor to this alignment shift barrel is a small addition.

7. Fault management

7.1 Introduction

Fault management, the process or processes by which errors and faults are isolated, logged, and mitigated, is the basis upon which highly reliable, maintainable systems are built (e.g., Futurebus+ Fault Tolerant Extensions, IEEE P896.9/D 2.5, May 1994). This section covers fault management aspects with respect to Futurebus+ and Futurebus+-based systems. Fault management concepts and terminology are first introduced setting the context upon which the following subclauses are based. These subclauses review error handling concepts, history and alternate access unit architectures, and bus-specific error descriptions.

Fault management is based upon a few very simple concepts. These concepts can produce systems that are robust enough to detect, analyze, and service errors as they occur. A subset of the fault management model that supports these design concepts is shown in figure 7. By building a hierarchical fault management architecture, the goal of continued system operation without data corruption, with minimal or no interruption, and with isolation of errors within a defined portion of the system can be achieved.

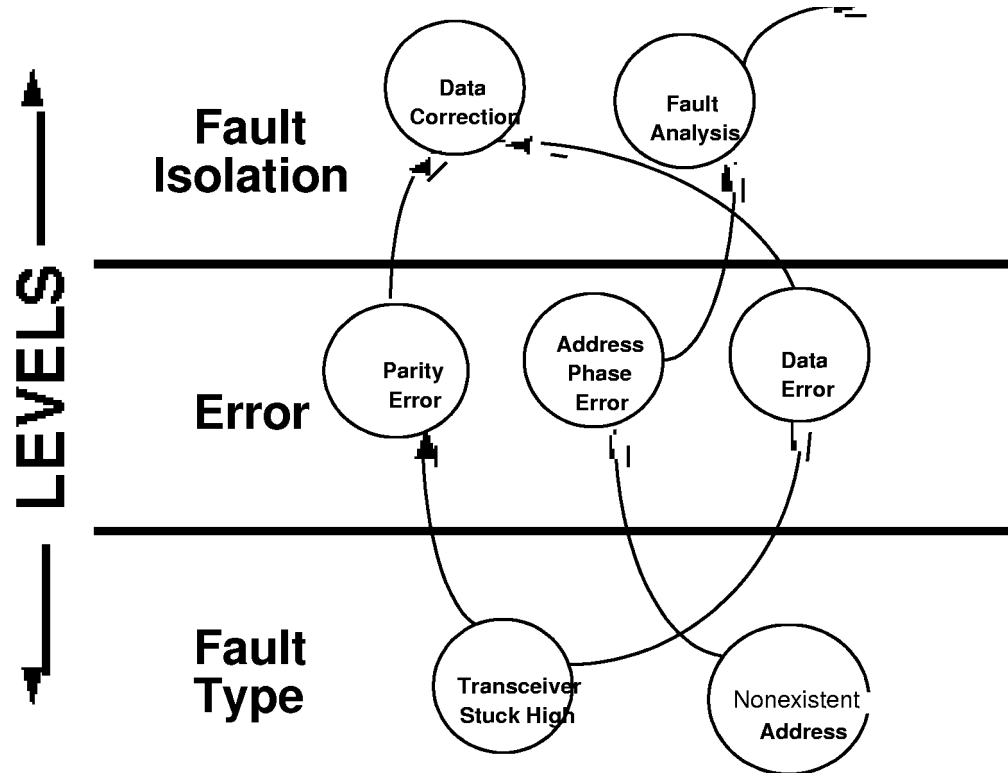


Figure 7— Fault management model

The process used to design fault management into a system starts with a thorough review of the requirements, followed by an analysis of the system design. The analysis should include a failure modes and effects analysis, reliability analysis, and system characterization. Fault management should be implemented to the lowest level possible in a system so as to gain the maximum diagnostic capability, recovery, and reporting effectiveness.

The basic concepts of fault management are as follows:

- Faults may cause errors. (Not all faults will cause an error, i.e., latent faults may exist and not cause an error in the system.)
- All errors are detected.
- Errors can be detected through proper design and placement of error detectors.
- Errors should be serviced at the lowest level possible.
- Error reports should always be available to a higher level.
- Errors may be selectively reported to a higher level.
- Faults should be mitigated at the lowest possible level.

This section assumes that the protocol and arbitration state machines are designed correctly and that fault management addresses hardware defects only. For topics related to designing the required degree of reliability and maintainability during hardware development, refer to section 8.

7.2 System fault management

System fault management is a collection of functions that when combined with a system configuration enables the timely management of the system in the presence of faults.

Fault management includes the following functions:

- a) *History unit.* Records system maintenance and state information for system maintenance.
- b) *Alternate access unit.* A means for isolating and mitigating faults.
- c) *Error handling.* Provides information for managing fault maintenance.

Error handling procedures, when utilized, can enable the system to continue operating in the presence of faults and provide the required information for managing fault maintenance throughout the life of the system. A history unit architecture provides the framework for recording pertinent information to support system maintenance throughout the system's life. An alternate access unit architecture defines the mechanism by which fault management processes can isolate and mitigate some system faults. Each of these areas when combined in a system provides the means for implementing a highly robust and cost-effective system.

7.2.1 History unit

A module's history of performance, revisions, and usage is important for minimizing the cost of system maintenance and for providing a resource for fault management. Many times system or card failures are dependent on past or current module stresses that only occur under specific circumstances. An adequate history log will reduce the cost of determining the cause and establishing the fix for these types of errors. A revision log will ensure that the present state of the module is known and interoperable. To support scheduled system maintenance, a module usage history log is desired. The following subclauses detail examples of how history unit architectures are implemented.

7.2.1.1 Module stress/Error history units

Nodes with processors can have a resident nonvolatile log that is used to provide node history and used as a buffer for storing error information. This log is active whenever it is in operation in a system. The log can be accessible (read or written) by way of the on-board diagnostic ability of the node, through the primary bus or through an alternate bus.

The log is optionally writable by way of an alternate bus for the purpose of adding in system-specific environmental information (e.g., unit temperature and altitude at the time of an error report) and for resetting logs. The data to be provided by the system may be broadcast to all nodes or requested by nodes per the system designer's approach. The log should be capable of storing a minimum of 64 stress/error entries.

The log is structured as shown in figure 8 below. The log provides pointers to the current and next available log entry. This is to minimize delays in fetching the current entry or writing the next entry in the log. If the log is full, the pointer should either wrap around, or continue to overwrite the last fault entry, based on a node configuration bit. Access to the stress/error log should be restricted to authorized processes. Erasure of log contents should be possible only through an alternate bus.

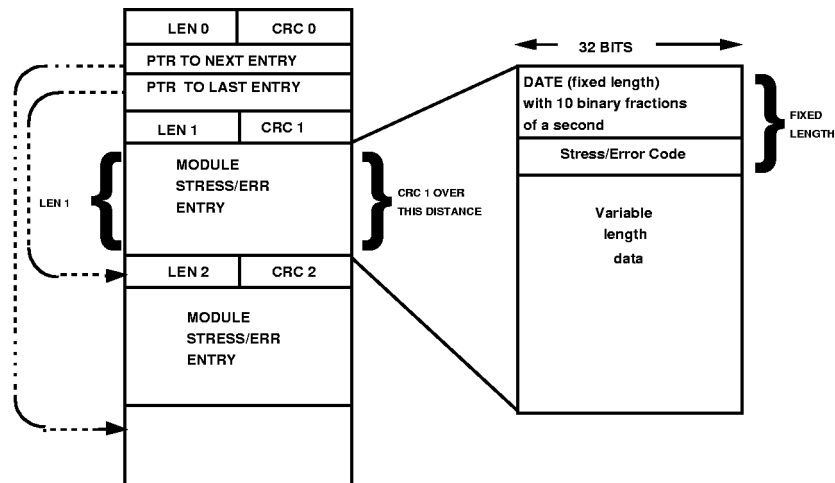


Figure 8— Module stress/History log

7.2.1.2 Module revision history unit

Modules can implement an EEPROM (or other nonvolatile memory) that contains a history of revisions to the module. The EEPROM should be able to store a minimum of 64 entries. A format for these entries is shown in figure 9 below.

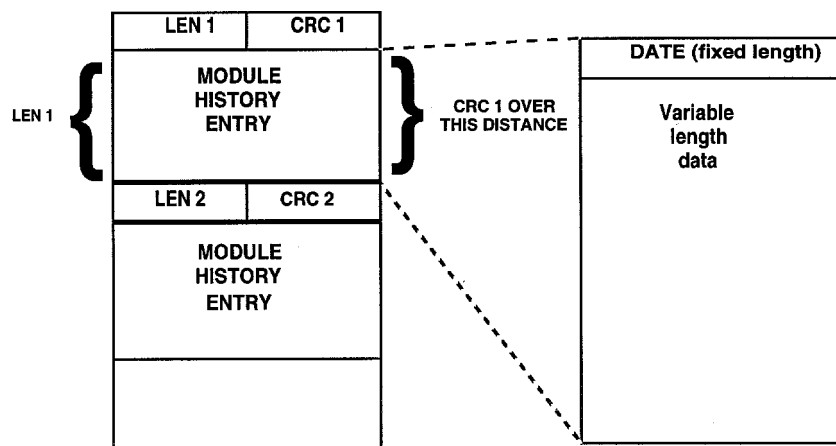


Figure 9— Module revision history log

Following these entries are pairs of entries that identify the revision keyed to the relevant change and the date of change.

7.2.1.3 Module usage history unit

All modules can implement an EEPROM (or other nonvolatile memory) that contains a history of usage of the module. The EEPROM should be able to store a minimum of 64 entries.

The first entry in this module is accumulated RUN TIME. This field is updated during the power down cycle, and adds the current power on time to the previous entry.

The second entry is a count of the module power on cycles. This count will be incremented during module initialization after power is determined to be stable.

The third and following entries are the location log. These are updated during module power on initialization. A system should have the ability to read the last entry in the usage history log. Based on that, the system should be able to determine if a module has been moved or installed and load a corresponding new entry into the log. Figure 10 illustrates this format.

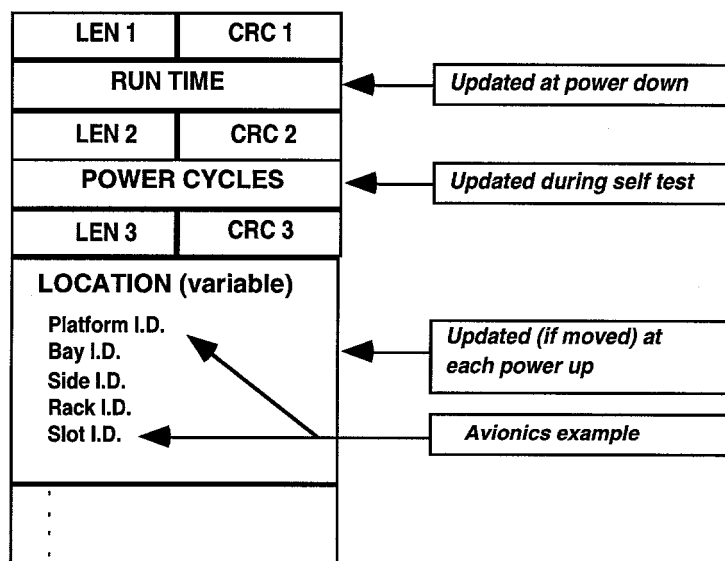


Figure 10— Usage/Location log

7.2.2 Alternate access unit (AAU)

An alternate access unit (AAU) is a type of unit architecture that defines access between multiple buses compliant to IEEE Std 1212-1991 (e.g., Futurebus+ and Serial Bus) when they share a common module. An AAU is simply a port into a portion of the local resources of the node sharing the module. An AAU should not be confused with a bridge, which has more capability and complexity. The AAU framework is general enough to allow for any number of buses compliant to IEEE Std 1212-1991 to access one another's CSRs and/or units. The purpose of this framework is to provide a general access method that will be used to specify individual AAUs. A full AAU architecture shall be specified for each type of AAU just as is the case for any other standard unit architecture.

There are two classes of AAUs—node and unit. Node AAUs specify alternate access to the CSR space of the nodes that share the module. Unit AAUs specify alternate access to a unit that is shared between nodes on the module.

A node AAU may be used when access to a node's CSRs is required from another bus. An example of this is the use of serial bus as a diagnostic path in a Futurebus+ system. The diagnostic path would require access to the CSRs of the Futurebus+'s node to activate module tests and to acquire test results. A unit AAU may be used when units need to be shared between buses. An example of a unit AAU is a module level stress/error history log. The log stores the fault history of the module, and needs to be accessible by each bus on the module.

The AAU framework specification makes restrictions on AAU units to simplify and standardize their implementation. The location of the AAU should be described in the unit's ROM space as per IEEE Std 1212-1991 ROM format specification. This is required of all standard units.

A node class AAU should be limited to the first 2 kb of CSR space as illustrated in figure 11. Units in the initial unit space may be accessed by using a separate Unit class AAU. A separate unit specification is required for each type of AAU unit. This is required of all standard units.

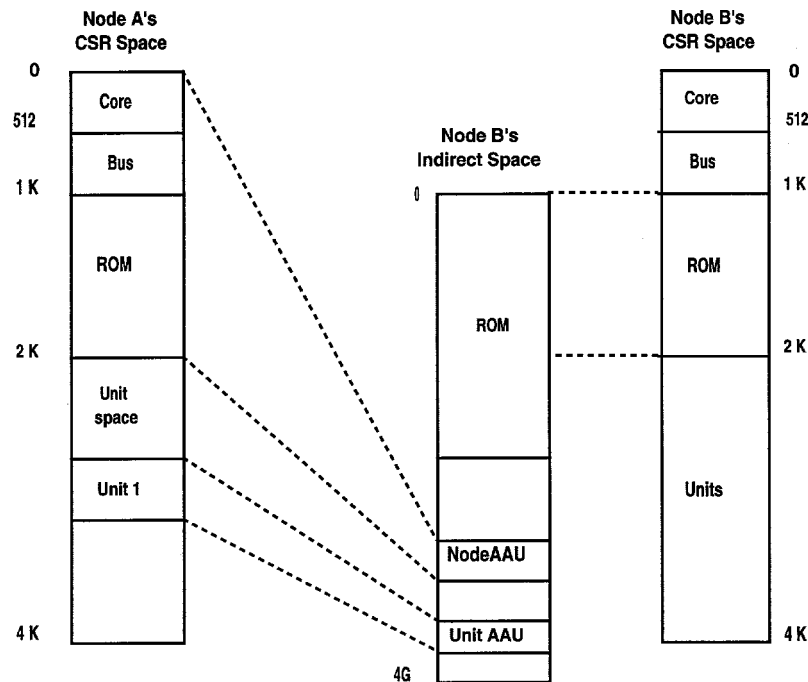


Figure 11— AAU mapping

The location of the AAU should be limited to the indirect space. To access the alternate node's CSR space, or units, the indirect_address and indirect_data register pair shall be used. However, the local access (between nodes on same module) is implementation specific and beyond the scope of this recommended practice.

The AAU access should be limited to module resources only, and should not cause or generate bus transactions.

7.2.3 Error handling

The arbitration and protocol sections of ISO/IEC 10857 : 1994 specify behavior of nodes on the bus in detailed state equations. The state equations identify errors that may be detected during different arbitration, connection, data, and disconnection phases. The behavior of the arbitration or protocol controller state machine is clearly specified. Conventions for recording state upon detection of errors, recovery by retry, and reporting errors to the next level in a system are not defined or specified in ISO/IEC 10857 : 1994. This subclause lists each error case separately with recommended practices for symptom capturing and signaling the error detection to the system.

All nodes should capture the same error symptoms. System bus bridges are expected to deviate in implementation of some of the conventions for signaling detection of an error on the bus. Unless otherwise specified, the following lexicon will be used in this subclause:

- *Master.* Module with the master attribute set
- *Slave.* Any module
- *Selected slave.* Module with the selected attribute set
- *Participating slave.* Module with the participating attribute set

7.2.3.1 Error strategy

Errors may be viewed from three dimensions:

- *Parallel protocol.* What status signals to assert in the event of an error.
- *Hardware level error symptom capturing.* What bits are set in the error symptom CSRs.
- *Error reporting.* How to notify a device driver or other higher-level entity in the system that an error has been detected.

Errors that shall be considered are the following:

- a) Connection phase
 - 1) Nonexistent address
 - 2) Address parity error
 - 3) Command parity error
 - 4) Capability error⁹
 - 5) Illegal operation error¹⁰
 - 6) Transaction time-out
 - 7) Retry threshold exceeded
 - 8) Nonexistent transaction ID or unexpected split response¹¹
- b) Data phase
 - 1) Data parity error
 - 2) Tag parity error
 - 3) Command parity error
 - 4) Transaction time-out
 - 5) Illegal operation error
- c) Disconnection phase
 - 1) Data parity error
 - 2) Tag parity error
 - 3) Command parity error
 - 4) Transaction time-out
 - 5) Length error
 - 6) Illegal operation error
- d) Node internal errors¹² and errors on the far side of a splitting bridge
- e) Distributed arbitration and arbitrated message bus errors
- f) Split time-out

Conditions and status signals that are not parallel protocol errors and will not be addressed are:¹³

- a) Wait status
- b) Intervention status
- c) Busy status
- d) Broadcast status
- e) Transaction flag (Conflict status)
- f) System initialization
 - 1) Power-up reset
 - 2) Bus initialize
 - 3) System reset
 - 4) Live insertion
 - 5) Power-related errors

⁹Operation is defined and not restricted in ISO/IEC 10857 : 1994, but is not supported by the module.

¹⁰Operations reserved in ISO/IEC 10857 : 1994. For example, assertion of **AW*** in a split response transaction is an illegal operation.

¹¹Not signaled on the bus. Reporting mechanism is unit or port-architecture specific.

¹²This may result in bad data on the bus.

¹³Assertion of these status signals may result in a bus-master-detected capability error.

7.2.3.2 Parallel protocol

This subclause attempts to explain the rules governing error strategy for Futurebus+ modules.

The following are general parallel protocol error signaling rules:

- a) Parallel protocol errors are signaled on the bus using **BE*** or **TE***. Any module may signal an error in the connection phase. A node with the participating attribute set may signal an error in all other phases. (ISO/IEC 10857 : 1994 indicates that a node may be master and have the participating attribute set at the same time. In this case it is possible for a node to be master and to drive these parallel protocol error status signals.)
- b) A master or participating slave that detects an error, or a master that receives a parallel protocol signal reporting that another module on the bus detected an error should set the appropriate bits in the ERROR_HI and ERROR_LO error log registers.
- c) Commands that are unsupported by a specific profile are treated with the same conventions as illegal operation errors. All other operations that are unsupported by a specific profile (e.g., those initiated by a signal on status lines or capability lines) are treated with the same conventions as capability errors. Capability and illegal operation errors both cause the PROTOCOL_ERROR bit to be set in the ERROR_HI CSR.

7.2.3.2.1 Master

A master may detect an error in a connected transaction or split transaction in any phase.

The master should save in the error log registers the command and address information from the connection phase, in the event that the transaction fails. The master should capture the state of the status and capability lines at the time it detects a parallel protocol error signal **BE*** or **TE***.

In all cases the master in a transaction should know about the error.¹⁴ It is the responsibility of the master to notify the requester that an error was detected. It is implementation dependent how a requester notifies the next level in the system. It is recommended that a unit-specific error interrupt be used to signal this event.

7.2.3.2.1.1 Connected transactions

On detection of an error or observation of a parallel protocol error signal **BE*** the master should proceed to the disconnection phase.

There is no time by which the standard requires a node to be in the disconnection phase after a **BE*** or **TE***. The master is permitted to continue handshakes and ignore data until the normal end of the data phase.

The master is the only frame of reference that can detect a TRANSACTION_TIMEOUT error. (A central arbiter can also detect this but there is no standard for what a central arbiter should do when it detects this condition.) In the event that the master detects a TRANSACTION_TIMEOUT the master will assert **RE*** to cause a BUS_INITIALIZE. In this case the master does not need to do anything extra to disconnect from the bus because asserting **RE*** will result in the initialization of the master state machine. (There seems to be no advantage to the MASTER disconnecting immediately in the case of a transaction time-out since, the immediate **RE*** assertion will prevent anyone else from using the bus in the 2 ms interval until BUS_INIT occurs.)

The master is the only frame of reference that can detect a RETRY_THRESHOLD_EXCEEDED error. (Retry threshold may be busy retry or error retry.) In the event that the error is a RETRY_THRESH_OLD_EXCEEDED, the master disconnects from the bus and does *not* drive any error status signal.

¹⁴One exception is slave-detected parity errors with parity error reporting disabled.