

IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses

Circuits and Devices

Communications Technology

IEEE Computer Society

Sponsored by the
Microprocessor and Microcomputer Subcommittee

*Electromagnetics and
Radiation*

Energy and Power

Industrial Applications

*Signals and
Applications*

*Standards
Coordinating
Committees*

IEEE Std 1212-1991



Published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.
July 28, 1992

SH15065

THIS PAGE WAS
BLANK IN THE ORIGINAL

IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses

Sponsor
**Microprocessor and Microcomputer Subcommittee
of the
IEEE Computer Society**

Approved December 5, 1991
IEEE Standards Board

Approved May 14, 1992
American National Standards Institute

Abstract: The document structure and notation are described, and the objectives and scope of the CSR Architecture are outlined. Transition set requirements, node addressing, node architectures, unit architectures, and CSR definitions are set forth. The ROM specification and bus standard requirements are covered.

Keywords: CSR Architecture, bus architecture, bus standard, interoperability, microprocessors, node addressing, registers, transaction sets

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

Copyright © 1992 by the
Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published in 1992
Printed in the United States of America

ISBN 1-55937-198-6

*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 which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least once 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 are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.



Documents

Document Sharing Hub

Foreword

(This foreword is not a part of IEEE Std 1212-1991, IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses.)

Bus standards have often been driven by hardware designers who have focused on the transport mechanisms for sending read and write transactions on a bus. Additional software considerations are needed to ensure interoperability between boards, as users of current bus “standards” have discovered. Therefore, many bus standards have been supplemented with one or several *de facto* or recommended register architectures, which have usually differed for each bus standard.

Through the cooperative efforts of the P1394 Serial Bus, P896 Futurebus+, and P1596 Scalable Coherent Interface (SCI) Working Groups, the need for a more formal approach to defining a common scalable bus-technology-independent Control and Status Register (CSR) Architecture was recognized. The hope is that, by sharing a uniform CSR Architecture, these systems will minimize the software and firmware changes when migrating a processor from one system bus to another or when bridging from one bus to another, and that software costs for migrating between standards (as technology evolves) will be reduced. The P1212 CSR Architecture Working Group was fortunate to have the wide range of bus technologies (from approximately 40 Mb/s for Serial Bus to approximately 1 Gbyte/s for SCI) to test the performance and cost scalability of its designs. The popularity of the Futurebus+ standard ensured that the CSR Architecture specification would be reviewed by a large audience for use in a wide variety of applications.

The scope of the CSR Architecture includes the definition of the generic registers needed to initialize, configure, and test nodes within a system. Other broadcast registers are sufficiently standardized to ensure interoperability between modules supplied by different vendors. The CSR document also defines address-space maps, bus transaction sets (reads, writes, and locks), and ROM data formats.

Protocols are defined for interrupting processors, passing messages, and for accurately synchronizing distributed clocks. These definitions are intended to provide a sufficient and standard framework for the design of vendor-dependent unit architectures. Parts of this CSR Architecture are likely to indirectly influence the processor designs of the future.

Acknowledgements

(These acknowledgements are not a part of IEEE Std 1212-1991, IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses.)

As a cross-functional development, the P1212 CSR Architecture Working Group was well endowed with experts from the P896, P1596, and P1394 working groups. The success of this effort was largely a function of these working group chairpersons, who encouraged their working group members to contribute to the process. They include Paul Borrill (Chair of IEEE Std 896.1-1991, IEEE Std 896.2-1991, P896.3), Michael Teener (Chair of P1394), and David Gustavson (Chair of IEEE Std 1596-1992).

The P1212 Chair was formally assisted in the development process by Dan O'Connor (who contributed as secretary and editor), Mike Wenzel (whose work on DMA is spawning a follow-up activity), Sam Duncan (who refined and documented the node-test protocols), and Dwight Wilcox (whose inputs drove the synchronized clock interface).

Many others also contributed to the task. As well as making many other contributions, Chet Pawlowski simplified the broadcast addressing structure and Jon Crowell simplified the ROM-access mechanism.

Don Senzig was an invaluable liaison between IEEE Std 1212-1991 and IEEE Std 896.2-1991, whose work was essential in the effort of making these two documents converge. Ed Jacques provided guidance and direction for the definition of the message-passing interface, which was based on concepts developed within IEEE Std 896.2-1991.

Our most active reviewers included Martin Whittaker, Mike Wenzel, and Ian Dobson. They, among others, provided extensive detailed review of the document drafts, within the short review periods that were provided.

Mark Bunker and Bill Evertz helped refine the synchronized clock interface. Their broad backgrounds helped to distinguish between the parts of the interface that should be standard and those that should remain bus- or vendor-dependent.

Harrison Beasley's support of the P1212 effort (while at the Navy) was a critical catalyst in giving credibility to the development of the standard. With his support, Ralph Lachenmaier and Mark Hassel were able to enunciate the Navy's interoperability concerns during the standards development process.

Ernst Kristiansen, Knut Alnes, and Marit Jenssen provided project inputs from the Scalable Coherent Interface side of the effort. Their flexibility in adapting quickly to the changes mandated by the other standards was invaluable for arriving at a consensus position.

Special thanks to Kurt Baty, who provided computer facilities and assistance for the final last-minute editing of the initial sponsor draft document. The members of the negative ballot review team (who have been previously acknowledged) helped produce this revised copy.

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

David V. James, Chair

Barbara Aichinger
Knut Alnes
Robert S. Baxter
Harrison Beasley
David Brash
Mark Bunker
Robert C. Carpenter
D. Del Corso
Jon Crowell
Stephen Deiss
Ian Dobson
Emer Dooley
Michael A. Dorsett
Sam Duncan
W.P. Evertz
Frank Fidducia
John R. Fortier
Ralph Frangioso

Tony Grigg
David B. Gustavson
Claes-Göran Gustavsson
Mark Hassel
David Hawley
Hubert Holierhoek
Ed Jacques
Marit Jenssen
Ernst Kristiansen
Ralph Lachenmaier
Jim Leahy
Jim McGrath
Thanos Mentzelopoulous
Jim Moidel
Klaus Mueller
George Nacht
Mitsunori Nakata
Richard Napolitano
Daniel C. O'Connor

Mira Pauker
Chet Pawlowski
James M. Pexa
Mike Roby
Tim Scott
Don Senzig
Patricia Smith
Joanne Spiller
Bob Squirrell
Haruhisa Suzuki
Micheal Teener
John Theus
Yoshiaki Wakimura
Mike Wenzel
Martin Whittaker
Hans Wiggers
Dwight Wilcox
Mark Williams
David L. Wright

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

John Allen
Knut Alnes
Harrison A. Beasley
Kyle M. Black
John Black
Kim Clohessy
Jonathan C. Crowell
Stephen L. Diamond
Samuel Duncan
Wayne Fischer
Joseph D. George
Andy Glew

David B. Gustavson
Zoltan R. Hunor
John Hyde
Ed Jacques
David V. James
Kenneth Jansen
Hubert Kirrmann
Ernst H. Kristiansen
Gerry Laws
James D. Mooney
Klaus Dieter Mueller
Gary A. Nelson
J. D. Nicoud

Daniel C. O'Connor
Mira Pauker
Donald Pavlovich
Richard Rawson
Carl Schmiedekamp
Don Senzig
Paul Sweazey
Michael G. Thompson
Eike Waltz
Hans A. Wiggers
Mark Williams
Oren Yuen



When the IEEE Standards Board approved this standard on December 5, 1991, it had the following membership:

Marco W. Migliaro, *Chair*

Donald C. Loughry, *Vice Chair*

Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Clyde Camp
James M. Daly
Donald C. Fleckenstein
Jay Forster*
David F. Franklin
Ingrid Fromm

Thomas L. Hannan
Donald N. Heirman
Kenneth D. Hendrix
John W. Horch
Ben C. Johnson
Ivor N. Knight
Joseph L. Koepfinger*
Irving Kolodny
Michael A. Lawler

John E. May, Jr.
Lawrence V. McCall
Donald T. Michael*
Stig L. Nilsson
John L. Rankine
Ronald H. Reimer
Gary S. Robinson
Terrance R. Whittemore

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Fernando Aldana
Satish K. Aggarwal
James Beall
Richard B. Engelman
Stanley Warshaw

Paula M. Kelty
IEEE Standards Project Editor

Contents

SECTION	PAGE
1. Document Structure and Notation.....	13
1.1 Document Structure	13
1.2 References.....	13
1.3 Conformance Levels.....	13
1.4 Technical Glossary.....	14
1.5 Bit, Byte, and Quadlet Ordering.....	20
1.6 Numerical Values	21
1.7 C Code Notation.....	21
1.8 CSR, ROM, and Field Notation	21
1.9 Register Specification Format.....	22
1.10 Reserved Registers and Fields.....	24
2. Objectives and Scope	25
2.1 Scope.....	25
2.2 Objectives.....	25
3. Transaction Set Requirements.....	27
3.1 Transaction Overview	27
3.2 Read and Write Transactions.....	27
3.3 Noncoherent Lock Transactions.....	28
3.4 Transaction Errors.....	30
3.5 Immediate Effects.....	31
4. Node Addressing	33
4.1 Node Addresses.....	33
4.2 Extended Addressing.....	33
4.3 64-Bit Fixed Addressing	35
4.4 Private Addresses.....	36
4.5 Initial Node Space.....	36
4.6 Extended Address Spaces	37
4.7 Indirect Space.....	38
4.8 Address Space Offsets	40
5. Node Architectures	42
5.1 Modules, Nodes, and Units	42
5.2 Node States.....	43
5.3 Node Testing.....	45
5.3.1 Access-Path Tests.....	45
5.3.2 Reset Test.....	45
5.3.3 Diagnostic Tests.....	45
5.3.4 Non-Standard Diagnostic Tests.....	47
5.4 Multinode Modules.....	47
5.5 On-line Replacement (OLR)	48
6. Unit Architectures.....	50
6.1 Unit Architecture Overview	50
6.2 Interrupts.....	50
6.2.1 Interrupt-Target Registers	50
6.2.2 Interrupt-Poll Registers.....	51
6.3 Message Passing.....	52
6.4 Globally Synchronized Clocks	52
6.4.1 Clock Overview	52
6.4.2 Clock Synchronization.....	53

SECTION	PAGE
6.4.3	Clock Update Models 54
6.4.4	Updating Clock Registers 55
6.4.5	Clock Accuracy Requirements 56
6.5	Memory Unit Architectures.....56
6.6	Unit Architecture Environment.....56
7.	CSR Definitions.....58
7.1	Register Names and Offsets.....58
7.2	Minimal Implementations 62
7.3	Unsupported Register Accesses.....62
7.4	Register Definitions 62
7.4.1	STATE_CLEAR.....62
7.4.2	STATE_SET.....66
7.4.3	NODE_IDS.....66
7.4.4	RESET_START.....68
7.4.5	INDIRECT_ADDRESS.....69
7.4.6	INDIRECT_DATA.....69
7.4.7	SPLIT_TIMEOUT.....70
7.4.8	ARGUMENT.....71
7.4.9	TEST_START.....72
7.4.10	TEST_STATUS.....75
7.4.11	UNITS_BASE.....79
7.4.12	UNITS_BOUND.....80
7.4.13	MEMORY_BASE.....81
7.4.14	MEMORY_BOUND.....83
7.4.15	INTERRUPT_TARGET.....83
7.4.16	INTERRUPT_MASK.....84
7.4.17	CLOCK_VALUE.....85
7.4.18	CLOCK_TICK_PERIOD.....86
7.4.19	CLOCK_STROBE_ARRIVED.....87
7.4.20	CLOCK_STROBE_INFO.....89
7.4.21	Message Targets.....89
7.4.22	ERROR_LOG Registers.....89
8.	ROM Specification.....91
8.1	Introduction.....91
8.1.1	ROM Design Assumptions.....91
8.1.2	ROM Formats.....91
8.1.3	Driver and Diagnostic Identifiers.....92
8.1.4	ASCII Text.....93
8.1.5	CRC Calculations.....94
8.2	ROM Formats.....96
8.2.1	First ROM Quadlet.....96
8.2.2	Minimal ROM Format.....96
8.2.3	General ROM Format.....96
8.2.4	Directory Formats.....98
8.2.5	Leaf Format.....101
8.2.6	Textual_Descriptor.....101
8.3	bus_info_block.....103
8.4	Root Directory Entries.....103
8.4.1	Bus_Dependent_Info.....104
8.4.2	Module_Vendor_Id.....104
8.4.3	Module_Hw_Version.....104

SECTION	PAGE
8.4.4 Module_Spec_Id.....	104
8.4.5 Module_Sw_Version	104
8.4.6 Module_Dependent_Info.....	105
8.4.7 Node_Vendor_Id.....	105
8.4.8 Node_Hw_Version	105
8.4.9 Node_Spec_Id.....	105
8.4.10 Node_Sw_Version	105
8.4.11 Node_Capabilities	105
8.4.12 Node_Unique_Id.....	106
8.4.13 Node_Units_Extent.....	107
8.4.14 Node_Memory_Extent	108
8.4.15 Node_Dependent_Info.....	109
8.4.16 Unit_Directory.....	109
8.5 Unit Directories	110
8.5.1 Unit_Spec_Id	110
8.5.2 Unit_Sw_Interface_Id.....	110
8.5.3 Unit_Dependent_Info	110
8.5.4 Unit_Location.....	110
8.5.5 Unit_Poll_Mask	111
8.6 Key Definitions.....	112
8.7 Company_Ids	113
9. Bus Standard Requirements.....	114
10. Bibliography.....	115

FIGURES	PAGE
Fig 1-1 Byte and Quadlet Ordering.....	20
Fig 1-2 CSR Format Specification (example).....	22
Fig 1-3 Unimplemented Register (hardwired to zero)	24
Fig 3-1 Simplified Lock Model	29
Fig 3-2 Responder CSR Processing Model.....	31
Fig 4-1 32-Bit Extended Addresses.....	33
Fig 4-2 64-Bit Extended Addresses.....	35
Fig 4-3 64-Bit Fixed Addressing	36
Fig 4-4 Initial Node Space Components	37
Fig 4-5 Configurable Extended Addresses.....	38
Fig 4-6 Indirect Space Mapping (address less than 1 Kbytes).....	39
Fig 4-7 Indirect Space Mapping (address greater or equal to 1 Kbytes)	39
Fig 4-8 ROM-Specified Offsets in the Initial Units Space.....	40
Fig 4-9 ROM-Specified Offsets in Extended Units Space.....	40
Fig 5-1 Simple Single-Bus System.....	42
Fig 5-2 Physical CSR Component Hierarchy.....	42
Fig 5-3 Node States	44
Fig 5-4 Initialization Test Interface.....	45
Fig 5-5 Diagnostic Test Interface	46
Fig 5-6 Disruptive Test Interface.....	46
Fig 5-7 Nondisruptive Test Interface.....	47
Fig 5-8 Multinode Module.....	48
Fig 5-9 OLR—Defective Module Removal	49
Fig 6-1 Polled CSR Interrupt Dispatch Model	47
Fig 6-2 Synchronized Node Clocks (broadcast backplane model).....	50
Fig 6-3 Synchronized Clocks (pipelined backplane model).....	50

FIGURES	PAGE
Fig 6-4 Synchronized Clock Registers	51
Fig 7-1 STATE_CLEAR Format	61
Fig 7-2 STATE_SET Format	65
Fig 7-3 NODE_IDS Format	66
Fig 7-4 RESET_START Format	68
Fig 7-5 INDIRECT_ADDRESS Format	69
Fig 7-6 INDIRECT_DATA Format	70
Fig 7-7 SPLIT_TIMEOUT Register	71
Fig 7-8 ARGUMENT Formats	71
Fig 7-9 ARGUMENT Register Remote Address Format	72
Fig 7-10 TEST_START Format	73
Fig 7-11 <i>cat</i> and <i>tops</i> Formats	73
Fig 7-12 TEST_STATUS Format	76
Fig 7-13 <i>test_state</i> Format	76
Fig 7-14 UNITS_BASE Formats	79
Fig 7-15 UNITS_BOUND Formats	80
Fig 7-16 MEMORY_BASE Formats	82
Fig 7-17 MEMORY_BOUND Formats	83
Fig 7-18 INTERRUPT_TARGET Format	84
Fig 7-19 INTERRUPT_MASK Format	85
Fig 7-20 CLOCK_VALUE Formats (read-only)	85
Fig 7-21 CLOCK_VALUE Formats (read/write)	86
Fig 7-22 CLOCK_TICK_PERIOD Formats	87
Fig 7-23 CLOCK_STROBE_ARRIVED Formats (read-only)	88
Fig 7-24 CLOCK_STROBE_ARRIVED Formats (read/write)	88
Fig 8-1 ROM Hierarchy	92
Fig 8-2 Software Identifier Hierarchy	93
Fig 8-3 First ROM Quadlet	96
Fig 8-4 Minimum ROM Implementation	96
Fig 8-5 Fully Implemented CSR ROM Directory	97
Fig 8-6 Directory Structure	98
Fig 8-7 Directory Entry Format	98
Fig 8-8 Immediate Entry Format	99
Fig 8-9 Offset Entry Format	99
Fig 8-10 Leaf or Directory Entry Format	99
Fig 8-11 Leaf Format	101
Fig 8-12 Textual Descriptor Locations	101
Fig 8-13 Textual Descriptor Leaf	102
Fig 8-14 Minimal ASCII Textual Descriptor Leaf Format	102
Fig 8-15 <i>bus_info_block</i> Format	103
Fig 8-16 Node Capabilities Entry Format	105
Fig 8-17 Unique_Id Leaf Format	106
Fig 8-18 Immediate Value Extent Entry Format	107
Fig 8-19 Node_Units_Extent CSR Format (offset format)	107
Fig 8-20 Unit_Locate Leaf Format	111
Fig 8-21 Unit_Poll Entry Format	111
TABLES	
Table 1-1 C Expression Summary	21
Table 1-2 CSR Addressing Conventions	22
Table 3-1 Required Transaction Types	27
Table 3-2 Optional Transaction Types	28

TABLES	PAGE
Table 3-3 Subcommand Values for Lock4 and Lock8	30
Table 3-4 CSRs with Special Split-Responder Effects	32
Table 4-1 32-Bit Extended Addresses	34
Table 4-2 64-Bit Extended Addresses	35
Table 5-1 Types of Node Reset	44
Table 7-1 CSR Locations	58
Table 7-2 CSR Access Modes	60
Table 7-3 CSRs Access Modes in Nonrunning States	61
Table 7-4 Minimal CSR Register Set	62
Table 7-5 Node State Values	65
Table 7-6 Special <i>test_step</i> values	74
Table 7-7 Expected Interpretation of <i>tops</i> Values	75
Table 7-8 Test Status Values	77
Table 7-9 Test Status Values (successful and unsuccessful phase)	78
Table 7-10 Test Status Values (active phase)	79
Table 7-11 Extended Space Alignment and Size	81
Table 8-1 ASCII Subset (number values are hexadecimal)	94
Table 8-2 CRC-16 Update Algorithm	95
Table 8-3 CRC-16 Calculation Routine	95
Table 8-4 <i>key_type</i> Definitions	98
Table 8-5 Calculating Address of Entry Value	100
Table 8-6 Fetching ROM From a Desired Offset Address	100
Table 8-7 Bus Name Example Values	103
Table 8-8 Root Directory Entries	104
Table 8-9 Node Capabilities Bit Field Definitions	106
Table 8-10 Immediate Value Units Extent Bit Field Definitions	107
Table 8-11 Pointer Value Node_Units_Extent Bit Field Definitions	108
Table 8-12 Extended Space Alignment and Size	108
Table 8-13 Immediate Memory_Extent Entry—Field Definitions	108
Table 8-14 Offset Memory_Extent Entry—Field Definitions	109
Table 8-15 Unit Directory Entries	110
Table 8-16 Unit_Locate.tag Bit Field Definition	111
Table 8-17 <i>key_type</i> and <i>key_value</i> (hexadecimal values)	112
APPENDIX	
A Guide for Using the CSR Architecture	117
A1. Bus Topologies	117
A1.1 Specialized Buses	117
A1.1.1 Multiple-Bus Topologies	117
A1.1.2 Dual-Port Nodes	118
A1.2 Fault Retry Protocols	119
A1.2.1 Hardware Fault Recovery	119
A1.2.2 Software Fault Recovery	119
A2. System Initialization	119
A2.1 System Initialization Summary	119
A2.2 Node Address Assignments	120
A2.3 Processor-Cache Model	121
A2.4 Address Protection	122
A2.5 Power Distribution Models	122
A3. Bus Transactions	124
A3.1 Transaction Overview	124
A3.2 Transaction Components	124

APPENDIX	PAGE
A3.3 Request Subaction Fields.....	125
A3.4 Response Subaction Fields.....	125
A4. Bus Bridges.....	126
A4.1 Address-Invariant Mappings	126
A4.2 Transaction Forwarding	127
A4.3 Transaction Ordering.....	128
A4.3.1 Split-Response Transaction Ordering.....	128
A4.3.2 Buffered-Write Transparency.....	128
A4.3.3 Weakly Ordered Move Transactions	130
A4.3.4 Queue-Dependency Deadlocks.....	130
A4.4 Address Domains.....	132
A4.5 Protection Boundaries.....	133
A4.6 Coherence Domains.....	134

APPENDIX FIGURES

Fig A1-1 Central Processor/Memory Bus.....	117
Fig A1-2 Multiple Bus Configurations	117
Fig A1-3 Serial Bus Extensions	118
Fig A1-4 Module With Redundant Bus Connections.....	118
Fig A2-1 node_id Address Assignments.....	120
Fig A2-2 Processor-Cache Model.....	121
Fig A2-3 TLBs Provide Selective User Protection	122
Fig A2-4 Power Distribution Model.....	123
Fig A3-1 Transaction Components	124
Fig A4-1 CSR Byte Ordering (big-endian)	126
Fig A4-2 Bridge Byte-Line Mapping (big- and little-addressian buses).....	127
Fig A4-3 Nontransparent Request Ordering.....	129
Fig A4-4 Nontransparent Response Ordering	129
Fig A4-5 Bridge Ordering—Shared Request/Response FIFOs Deadlock	130
Fig A4-6 Deadlocked Multiprocessor Interrupts	131
Fig A4-7 Deadlocked Message Queues.....	131
Fig A4-8 Two Addressing Domains.....	132
Fig A4-9 32/64-Bit Bridge Mapping—Translations and Conversions.....	133
Fig A4-10 Remote Subaction Checking	134
Fig A4-11 Limited Coherence Domain	135

IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses

1. Document Structure and Notation

1.1 Document Structure. The CSR Architecture is split into two components, as listed below:

IEEE Std 1212 CSR Architecture

The specification portion of this document defines the address-space maps, the bus transaction sets, and the node's CSRs. The intention is to provide a sufficient and standard framework for the design of vendor-dependent unit architectures.

The specification includes the format and content of the configuration ROM on the node. The configuration ROM provide the parameters necessary to auto-configure systems with nonprocessor nodes provided by multiple vendors.

Note that a monarch selection process, which selects one processor to boot the system, is not defined. A monarch selection process would be necessary to initialize a system containing processors provided by different vendors.

Appendix: A Guide for Using the CSR Architecture

The appendix provides background for understanding the usage of this CSR Architecture specification.

The CSR Architecture provides the specification upon which conforming designs should be based. The appendix illustrates ways that these capabilities could be used. Note that the appendix is nonbinding.

1.2 References. This standard shall be used in conjunction with the following publication. When it is superseded by an approved revision, the revision shall apply:

[1] ISO 646:1983, Information processing—ISO 7-bit coded character set for information interchange.¹

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

expected. A key word used to describe the behavior of the hardware or software in the design models assumed by the CSR Architecture. Other hardware and software design models can also be implemented.

may. A key word that indicates flexibility of choice with no implied preference.

shall. A key word indicating a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other CSR Architecture conformant products.

should. A key word indicating flexibility of choice with a strongly preferred alternative. The phrase *it is recommended* has the same meaning.

¹ ISO documents are available from ISO Central Secretariat, 1 rue de Varembe, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse; and from the Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018, USA.

1.4 Technical Glossary. A large number of bus and interconnect-related technical terms are used in the CSR Architecture document. These terms are described herein:

active test. An on-going test that is invoked by a write to the TEST_START register. The node is in the testing state (STATE_CLEAR.state is equal to testing) while an active test is in progress.

address_error. An error-status code returned to the requester when a transaction is directed to a non-existing address; on some buses, this has been called a NACK (negative acknowledge). The address_error status is generally returned if a valid address acknowledgement is not observed within a fixed timeout period.

addressing. *See:* extended addressing (32-bit); extended addressing (64-bit); fixed addressing (64-bit).

agent. An active switch, switch-like component, or bridge, between the requester and responder. During normal system operation, the agent is transparent to the requester and responder.

backplane. A subassembly that holds the connectors into which one or more boards can be plugged. In addition to providing bus signal connections, the backplane usually provides power connections, power status information, and physical position information to the board.

big addressian. A term used to describe the physical location of data-byte addresses on a multiplexed address/data bus. On a big-addressian bus, the data byte with the largest address is multiplexed (in time or space) with the least-significant byte of the address.

big endian. A term used to describe the arithmetic significance of data-byte addresses within a multibyte register. Within a big endian register or register set, the data byte with the largest address is the least significant.

board. The physical component that is inserted into one of the backplane slots.

bridge. A hardware adapter that forwards transactions between buses.

broadcast transaction. A transaction that is distributed to all nodes on a bus.

buffered write. A write transaction that appears to complete when the request is queued in the agent or responder. A buffered-write transaction returns an optimistic (done_correct) status before the responder's completion status (which could report an error) is available.

bus-dependent. A term used to describe parameters that may vary among different bus standards, but are defined by them. Although the CSR Architecture may constrain the definition of these fields, their detailed definition is provided by the appropriate bus standard.

bus standard. An abbreviated notation used throughout this document, rather than the more exact "bus standard document that claims conformance to this specification."

byte. A byte is 8 bits of data.

coherent transaction. A transaction (typically read or write) that provides protocols for checking and maintaining consistency with other caches. Coherent transactions are expected to address a cache-line. For example, tightly coupled multiprocessors are expected to use coherent transactions when accessing shared-memory resident data.

command_reset. An initialization event that is initiated by a write to the RESET_START register.

company_id. A 24-bit binary value used to identify a company within the context of the CSR Architecture. The company_id values are expected to be uniquely assigned to each company.

conflict_error. An error-status code that is returned when a transaction has been transmitted successfully, but a queue or usage conflict inhibits the transaction completion. A `conflict_error` status is returned to the original requester, which is expected to retry the transaction. This is different than a bus-dependent delay (wait or busy status), which delays the forwarding of a transaction or subaction across the bus.

CSR. An abbreviation for control and status register. A CSR is a quadlet register that is accessed through `read4` or `write4` transactions and is used to observe a node's state or to control its operation.

CSR Architecture. A term that refers to this standards document.

dead state. A node state that is reflected by the value of 3 in the `STATE_CLEAR.state` field. A node enters the dead state when a fatal error has been detected and the node is connected but no longer operational. Note that the severest errors could leave the node in a broken state, with its registers undefined, rather than indicating a dead state.

diagnostic test. A test, or collection of tests, that is invoked by writing to the `TEST_START` register. There are four forms of diagnostic tests: initialization tests, extended tests, manual tests, and system tests.

directory. A contiguous collection of one or more entries, which is contained within the node's ROM.

directory entry. A ROM entry that specifies the address of another ROM directory.

disconnected state. A state in which the node no longer responds to bus transactions. Since the node no longer responds to bus transactions, a `power_reset` is required to change to another node state.

disruptive test. A test that is invoked through a write to the `TEST_START` register and disrupts the node's operation by temporarily moving the node to the testing state.

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

done_correct. A status code that is returned when a transaction is completed without errors. On many buses, the `done_correct` status is implicitly assumed when no error-status codes are observed.

doublet. Two bytes (16 bits) of data.

emperor processor. The monarch processor that is selected to initialize and configure the system. On a single-bus system, the monarch and emperor processor are always the same. On a multiple-bus system, the single emperor processor is selected from the available monarch processors.

entry. (1) *See: ROM.* (2) A component of a directory, which is located within the node's ROM. An entry may contain information, or a pointer to another directory or leaf.

extended addressing (32-bit). The address model implemented by bus standards supporting 32-bit addresses. The 32-bit extended addressing model is a subset of the 64-bit extended addressing model.

extended addressing (64-bit). An address model implemented by bus standards supporting 64-bit and 32-bit addresses. The 32-bit extended addressing model is a subset of the 64-bit extended addressing model.

extended memory space. An extended address space on a node that provides a RAM-access window for a memory-controller unit architecture. The base address and upper bound of

the extended memory space are specified through writes to the node's MEMORY_BASE and MEMORY_BOUND registers. The extended memory space is not relevant to bus standards implementing 64-bit fixed addressing.

extended test. A test or collection of tests that shall perform bus transactions and use an external memory buffer. An extended test is invoked by writing to the TEST_START register.

extended units space. An extended address space on a node that provides an access window for unit architectures. The base address and upper bound of the extended units space are specified through writes to the node's UNITS_BASE and UNITS_BOUND registers. The extended units space is not relevant to bus standards implementing 64-bit fixed addressing.

fixed addressing (64-bit). An address model implemented by bus standards supporting only 64-bit addresses. The initial node space is large (258 Tbytes), is fixed in size, and extended spaces are not supported.

Futurebus+. A name that refers to IEEE 896.1-1991 [B3]² and companion 896.x standards. Futurebus+ defines a physically bussed backplane bus standard, which supports 32-bit and 64-bit physical addresses.

general ROM format. A format for the node-provided ROM. The general ROM format provides bus-dependent information and a root_directory; the root_directory directly provides additional ROM entries.

immediate effect. An effect of a transaction, which appears to occur between the time the request subaction is accepted and the response subaction is returned. If a bus standard allows CSR transactions to be split, and sufficient time is allowed between the acceptance of a request subaction and the return of a response subaction, an immediate effect can be emulated by a processor on the node.

immediate entry. A ROM entry that provides a 24-bit immediate data value.

independently powered. An adjective used to describe a node on a bus, when the node's power supply may fail while other nodes remain powered and operational.

initial memory space. A portion of the initial node space, which provides a RAM-access window for a memory-controller unit architecture. Unit architectures can also be mapped to non-conflicting portions of the initial memory space. The initial memory space is only relevant to bus standards implementing 64-bit fixed addressing.

initial node space. The address space that is initially mapped to a node. The initial node space contains the initial register space (2 Kbytes) and initial units space. On buses implementing 32-bit or 64-bit extended addressing, the initial units space is 4 Kbytes in size. On buses implementing 64-bit fixed addressing, the initial node space is 256 Tbytes in size and also includes the node's initial memory and private spaces.

initial register space. A 2-Kbyte portion of the initial node space that is adjacent to the initial units space. The registers defined by the CSR Architecture are located within the initial register space. The initial register space also provides addresses for defining bus-dependent registers.

initial units space. The portion of the initial node space that is adjacent to but above the initial register space. When its size is sufficient, unit architectures are expected to be located within this space.

² The numbers in brackets refer to those of the references in 1.2; when preceded by a "B" they refer to those of the Bibliography in Section 10.

initialization test. A test or collection of tests that does not require cooperation of any other node(s) on the bus. The default and vendor-dependent initialization tests are invoked by writing to the TEST_START register.

initializing state. A node state that is reflected by the value of 1 in the STATE_CLEAR.state field. The initializing state is an optional transient state which is entered immediately after a power_reset or command_reset event.

Kbyte. Kilobyte. Indicates 2^{10} bytes.

leaf. A contiguous information field that is pointed to by a ROM-directory entry. A leaf contains a header (length and CRC specification) and other information fields.

leaf entry. A ROM entry that specifies the address of a leaf.

line. An aligned block of data on which coherence checks are performed. Several bus standards use 64-byte lines, so it is the expected line size for the CSR Architecture (although other line sizes are not prohibited). For example, the 64-byte line size is reflected in the size of bus transactions used for message passing.

little addressian. A term used to describe the physical location of data-byte addresses on a multiplexed address/data bus. On a little addressian bus, the data byte with the smallest address is multiplexed (in time or space) with the least-significant byte of the address.

little endian. A term used to describe the arithmetic significance of data-byte addresses within a multibyte register. Within a little endian register or register set, the data byte with the smallest address is the least significant.

lock transaction. A transaction that passes an address, subcommand, and data parameter(s) from the requester to the responder and returns a data value from the responder to the requester. The subcommand specifies which indivisible update is performed at the responder; the returned data value is the previous value of the updated data.

manual test. A test or collection of tests that requires an operator. The tests may involve power failure testing, on-line replacement testing, media testing, or cable connection tests (when loop-back cables are required). A manual test is invoked by writing to the TEST_START register.

Mbyte. Megabyte. Indicates 2^{20} bytes.

minimal ROM format. A format for the node-provided ROM. The minimal ROM format provides a 24-bit company_id value; although additional ROM parameters can be provided, their format and meaning are vendor-dependent.

module. A board, or board set, consisting of one or more nodes that share a physical interface. Although only one board in a module connects to bus signals, each board connector could provide power from the bus.

monarch processor. The processor that is selected to partially initialize the local-bus resources and fetch the initial boot code.

node. The software-visible station on a bus. Each node is allocated a set of control register addresses (including identification-ROM and reset command registers) that are initially defined in a 4-Kbyte (minimum) initial node address space. Although multiple nodes may share one bus interface, each node can be reset independently (a reset of one node has no effect on other nodes).

nodecast. An adjective used to describe an interrupt or message transaction that is distributed to all units on a node. Also used as a verb; e.g., "transactions may be nodecast to all units on a node."

node_id. A 16-bit value that determines the initial node address space. On some buses, the *node_id* value has two components: a *bus_id* field specifies one of 1024 bus address and an *offset_id* field specifies one of 64 node positions on the bus. During system initialization, software is expected to assign unique *node_id* values to nodes within a system.

noncoherent transaction. A transaction (typically read or write) that is completed without checking for consistency with other caches on the local bus. For example, noncoherent 4-byte read and write transactions are used to access CSRs. Note that noncoherent transactions may be converted into coherent transaction sequences when passing through bus bridges.

nondisruptive test. A test that is invoked through a write to the *TEST_START* register and does not disrupt the node's operation (the node does not enter the testing state).

nonvolatile memory (NVM). Read/write storage that is preserved through losses of power.

offset entry. A ROM entry that provides a 24-bit offset value. The offset value specifies the location of a CSR that provides a 32-bit parameter.

physical interface. The circuitry that interfaces the module, board(s), and node(s) to the bus signals.

power_reset. An initialization event triggered by the restoration of primary power. On a backplane bus, a *power_reset* event is generally triggered by one or several specialized signals driven by the shared power supply.

primary state. The state on a node that is initialized by a *power_reset*. For example, the CSRs defined by IEEE Std 1212-1991, CSR Architecture, are part of the node's primary state.

quadlet. Four bytes (32 bits) of data.

read transaction. A transaction that passes an address and size parameter from the requester to the responder and returns data values from the responder to the requester. The size parameter specifies the number of bytes that are transferred.

register. A term used to describe quadlet addresses that can be read or written by software. In the context of this document, a register does not imply a specific hardware implementation. If a bus standard allows transactions to be split, and sufficient time is allowed between the request and response subactions, the functionality of the register can be emulated by a processor on the module.

request subaction. A subaction that is generated by a requester to initiate an action on the responder. For a processor-to-memory read transaction, for example, the request subaction transfers the memory address and command from the processor to memory.

requester. A node that initiates a transaction by generating a request subaction (containing address, command, and sometimes data).

reset test. A test or collection of tests that is invoked by a *command_reset*. Although a reset test is actually a form of initialization test, the term reset test is used to avoid confusing its functionality with the initialization tests that are invoked by writing to the *TEST_START* register.

responder. A node that completes a transaction by returning a response subaction (containing completion status and sometimes data).

response subaction. A subaction that is returned by a responder to complete a transaction initiated by a requester. In a processor-memory read transaction, for example, the response subaction returns the data and status from the memory to the processor.

response_timeout. An implied split-transaction-error status that is returned when the response subaction is not returned within an expected timeout interval.

ROM. An abbreviation for read-only memory. The ROM data is maintained through losses of power. In some implementations ROM may actually be writeable, using (normally disabled) vendor-dependent protocols.

root_directory. A term used to describe the directory at the top level of the hierarchical ROM directory structure.

running state. A node state that is reflected by the value of 0 in the `STATE_CLEAR.state` field. The running state is the normal operational state in which access to all of the node's CSRs are defined.

SCI (Scalable Coherent Interface). The name that refers to IEEE Std 1596-1992 [B9]. Though functionally behaving as a bus, the SCI's physical implementation is a collection of point-to-point unidirectional links (i.e., a ring).

Serial Bus. The name that refers to the IEEE project, P1394 [B8], which specifies a serial bus intended as a low-cost peripheral connect or an alternate diagnostic and control path.

split transaction. A transaction that consists of separate request and response subactions. On a backplane bus, for example, bus ownership is relinquished between the request and response subactions. A transaction that is not split is called a unified transaction.

standard bus. An abbreviated notation used throughout this document, rather than the more exact "physical bus standard that claims conformance to this specification."

subaction. One of the two components in a transaction; a transaction consists of request and response subactions.

subunit. A logical subcomponent of a unit that is accessed by a largely independent subcomponent of I/O driver software. For example, a terminal multiplexer unit could have multiple subunits (two for each full-duplex connection).

successful test. A completed test that is invoked by a write to the `TEST_START` register, in which no errors are detected.

supported transaction. A transaction whose returned data value and side effects are defined by the hardware architecture that is addressed. For example, a write4 transactions to the 4-byte `STATE_CLEAR` register is supported.

system test. A test, or collection of tests, that may use an external memory buffer and may require cooperation of other nodes. For example, identical unit architectures may collaborate to test cache coherence or to generate background "noise" traffic for other nodes being tested. A system test is invoked by writing to the `TEST_START` register.

Tbyte. Terabyte. Indicates 2^{40} bytes.

testing state. A node state that is reflected by the value of 2 in the `STATE_CLEAR.state` field. The testing state is an optional transient state that is entered immediately after a write to the `TEST_START` register. The node remains in the testing state until the active test completes.

transaction. A transfer between requester and responder consisting of a request and response subaction. The request subaction transfers a command (and sometimes data) between a requester and responder. The response subaction returns status (and sometimes data) from the responder to the requester. A transaction may be either unified or split.

type_error. A status code that is returned when the transaction is directed to an existing address, but the transaction command (for example, a read64 directed to a quadlet register) is not implemented.

unified transaction. A transaction in which the request and response subactions are completed as an indivisible sequence. Between the initiation of the request and the completion

of the response, other subactions are blocked. The Futurebus+ standard also calls this a connected transaction. A transaction that is not unified is called a split transaction.

unit. A subcomponent of the node that provides a processing, memory, or I/O functionality. After the node has been initialized (typically by generic software), the unit provides the register interface that is accessed by I/O driver software. The units normally operate independently of each other, and do not affect the operation of the node upon which they reside. Note that one node could have multiple units (e.g., processor, memory, and SCSI controller).

unit architecture. The specification document describing the format and function of the unit's software-visible registers.

unit-dependent. A term used to describe parameters that may vary between different unit architectures. Although the CSR Architecture may specify the size and location of these fields, their format and most of their definition is provided by the appropriate unit architecture specification.

unsuccessful test. A completed test that is invoked by a write to the TEST_START register and detects one or more errors.

unsupported transaction. A transaction whose returned data value or side effects are not defined by the hardware architecture that is addressed. For example, a write64 transactions to the 4-byte STATE_CLEAR register is unsupported.

vendor-dependent. A term used to describe parameters that may vary between vendors supplying the same node or unit architectures. Although the CSR Architecture may constrain the definition of these fields, their format and definition is provided by the module vendor. Note that vendor-dependent fields may be standardized or left implementation-specific, depending on the vendor's needs.

write transaction. A transaction that passes an address, size parameter, and data values from the requester to the responder. The size parameter specifies the number of bytes that are transferred.

1.5 Bit, Byte, and Quadlet Ordering. The CSR Architecture defines registers that are 4 bytes (or larger) in size. To ensure interoperability across bus standards, the ordering of the bytes within these registers is defined by their relative addresses, not their physical position on the bus. Bus bridges are similarly expected to route data bytes from one bus to another based on their addresses, not their physical position on a bus. The routing of data bytes based on their address is called address-invariance.

To support the address-invariance model, bus standards shall specify the mapping of their physical byte-lanes to relative data-byte addresses. The CSR Architecture specifies the mapping of data-byte addresses to bytes within the multibyte CSRs. These two specifications indirectly specify the mapping between a CSR and physical byte-lanes on the bus. For a quadlet CSR access, the data byte with the smallest address is the most significant, as illustrated in Fig 1-1.

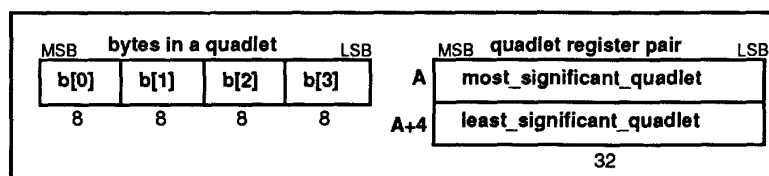


Fig 1-1
Byte and Quadlet Ordering

Since 64-bit addressing is supported throughout this standard, many values are stored as quadlet register pairs. For consistency, the quadlet register with the smallest address is also the most significant, as illustrated in Fig 1-1. Note that different byte-ordering conventions may be applied to the vendor-defined unit-dependent registers. For example, a graphics frame buffer could route data-byte-zero to the least-significant portion of a pixel-depth parameter. These unit-dependent formats are beyond the scope of the CSR Architecture.

For most of the CSRs, the size of all fields within the quadlet are specified; the bit position of each field is implied by the size of fields to its right or left. This labeling convention is more compact than bit-position labels, and avoids the question of whether 0 should be used to label the most or least-significant bit.

1.6 Numerical Values. Decimal, hexadecimal, and binary numbers are used within this document. For clarity, the decimal numbers are generally used to represent counts, hexadecimal numbers are used to represent addresses, and binary numbers are used to describe bit patterns within binary fields.

Decimal numbers are represented in their standard 0, 1, 2, . . . format. Hexadecimal numbers are represented by a string of one or more hexadecimal (0–9,A–F) digits followed by the subscript 16. Binary numbers are represented by a string of one or more binary (0,1) digits followed by the subscript 2. Thus the decimal number “26” may also be represented as “1A₁₆” or “11010₂”.

1.7 C Code Notation. The meaning of several arithmetic algorithms are formally defined by C code. Since many of the C code operators are not obvious to the casual reader, their meanings are summarized in Table 1-1. If the case of conflict, the definition in ANSI X3.159-1989 [B1] takes precedence.

Table 1-1
C Expression Summary

Expression	Description
$\sim c$	Bitwise complement of integer c
$c \wedge d$	Bitwise EXOR of integers c and d
$c \& d$	Bitwise AND of integers c and d
$c d$	Bitwise OR of integers c and d
$c << d$	Integer c , shifted left by d bits
$c >> d$	Integer c , shifted right by d bits
$c == d$	Boolean, true if integer c is equal to integer d
$c != d$	Boolean, true if integer c is not equal to integer d
$!b$	Logical negation of Boolean b
$a \&\&b$	Logical AND of Booleans a and b
$a b$	Logical OR of Booleans a and b

1.8 CSR, ROM, and Field Notation. This document describes a large number of control and status registers (CSRs) and fields within these registers. To distinguish register and field names from node states or descriptive text, the register name is always capitalized and the field name is always in *italics*. Thus, the notation **STATE_CLEAR***.lost* is used to describe the *lost* bit within the **STATE_CLEAR** register. In this paragraph, as well as the rest of this document, a bold type font is used to emphasize a term, particularly on its first usage.

All CSRs are quadlets and are quadlet-aligned. The address of a register (which is always a multiple of 4) is specified as the byte-offset from the beginning of the initial node space. When a range of register addresses is described, the ending address is the address of the last register, which is also a multiple of 4. These addressing conventions are illustrated in Table 1-2.

Table 1-2
CSR Addressing Conventions

Offset	Register Name	Description
0	STATE_CLEAR	First CSR location
4-12	OTHER_REGISTERS	Next three CSR locations

When a CSR register is defined, the address represents an address offset from the beginning of the initial node space, which is described in a later section.

This document describes a large number of ROM entries and fields within these entries. To distinguish ROM entry and field names from node states or descriptive text, the first character of the entry name is always capitalized and the field name is always in *italics*. Thus, the notation **Node_Memory_Extent***.align* is used to describe the *align* bit within the **Node_Memory_Extent** entry.

1.9 Register Specification Format. This document precisely defines the format and function of CSRs. Some of these registers are read-only, some are read/write, and some have special side effects on writes. To accurately define the content and function of these CSRs, their specification includes the format (the sizes and names of bit field locations), the initial value of the register (if not zero), the value returned when the register is read, and the effect(s) when the register is written. An example register is illustrated in Fig 1-2.

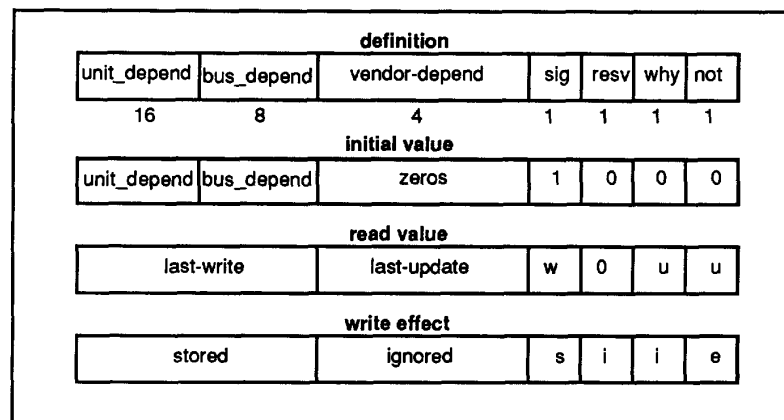


Fig 1-2
CSR Format Specification (example)

The fields within a register are named in the register's definition; these names are somewhat descriptive, but may be too short to be meaningful. The meaning of these fields is

defined in the text; their capabilities should not be implied by their names. However, the following register-definition fields have generic meanings:

unit_depend:

Definition The meaning of this field shall be defined by the node's unit architecture(s).

bus_depend:

Definition The meaning of this field shall be defined by the bus standard.

vendor_depend:

Definition The meaning of this field shall be defined by the node's vendor.

Within a unit architecture, the *unit_dependent* fields may be defined to be *vendor_dependent*. Within a bus standard, the *bus_dependent* fields may be defined to be *vendor_dependent*.

The node registers defined in the CSR Architecture shall be initialized when power is restored (a **power_reset**) or when a quadlet is written to its **RESET_START** register (a **command_reset**). For most registers, the initial value after a **power_reset** or **command_reset** is the same. When the initial CSR values differ, the two initial values are explicitly illustrated.

State in the CSRs need not be lost when the CSRs are initialized (in the sense that the previous values are irretrievable). An implementation may save the CSR contents in non-volatile memory before power is lost; bus standards are expected to provide a power-failure warning, to support such state-save protocols. After a **power_reset** or **command_reset** initializes the CSRs to (possibly) new values, the previously saved CSR values may be recalled using bus- or vendor-dependent mechanisms.

The following read-value fields have a generic meaning:

last_write:

Abbreviation *w*
Definition The value of the data field shall be the value which was previously written to the same register address.

last_update:

Abbreviation *u*
Definition The value of the data field shall be the last value which was updated by node hardware.

The following write-effect fields have a generic meaning:

stored:

Abbreviation *s*
Definition The value of the written data field shall be immediately visible to reads of the same register.

ignored:

Abbreviation *i*
Definition The value of the written data field shall be ignored; it shall have no effect on the node's state.

effect:

Abbreviation *e*
Definition The value of the written data field shall have an effect on the node's state, but is not immediately visible to reads of the same register.

The register description specifies the register's optionality (required or optional) as well as its read/write characteristics. A read/write register (**RW**) is expected to be read and written by software; a read-only register (**RO**) is expected to be only read by software; a write-only register (**WO**) is expected to be only written by software. Although reads and writes of **WO** or **RO** registers are not expected, the register definition still defines their results.

1.10 Reserved Registers and Fields. Some of the CSR addresses correspond to unimplemented registers. This includes optional registers (when the option is not implemented) and reserved registers (which are required to be unimplemented). The capabilities of these unimplemented registers are exactly defined to minimize conflicts between current implementations and future definitions, as illustrated in Fig 1-3.

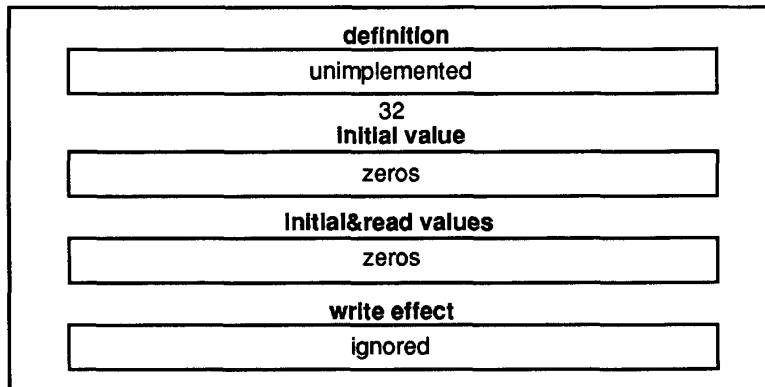


Fig 1-3
Unimplemented Register (hardwired to zero)

Within an implemented register, a field that is reserved for future revisions of this standard is labelled *reserved* (sometimes abbreviated as *r* or *resv*). For a reserved field within an implemented register, the field is ignored on a write and returns zero on a read, as formally specified below:

reserved:

Required	Reserved for future definitions.
Initial Value	Zero.
Read4 Value	Shall return zero.
Write4 Effect	Shall be ignored.

2. Objectives and Scope

2.1 Scope. This section summarizes the feature sets provided by the CSR Architecture and illustrates how these features are expected to be used. The CSR Architecture supports the concept of bus bridges, which (after being properly initialized) can transparently forward transactions from one compliant bus to another. This simplifies software development and encourages the use of specialized (low-cost or high-performance) bus standards. By defining a common CSR Architecture for multiple buses, the amount of customized software necessary to support each bus standard is minimized.

To improve the amount of software transparency in such multiple-bus configurations, the scope of the CSR specification includes the following:

- (1) *Physical Address Space Partitions.* The partitioning of the address space between node CSRs and memory is defined. Both 32-bit and 64-bit addressing options are allowed.
- (2) *Common Transaction Sets.* A common transaction set (including error-status codes) is defined. This transaction set can be transparently passed through bridges.
- (3) *Core CSRs.* The location and meaning of the core CSRs, which are accessed during the system initialization process, are defined. This provides a uniform software interface, independent of the physical bus location.
- (4) *ID-ROM.* The format and meaning of the node's ROM data structures are defined. The ROM directory structure supports standard and vendor-dependent data types.
- (5) *Interrupts.* Standard target addresses are provided for interrupts that are broadcast or nodecast to multiple nodes. Other vendor-dependent quadlet registers may be provided for interrupts that are directed to individual units.
- (6) *Messages.* Standard target addresses are provided for messages that can be broadcast or nodecast to multiple nodes. Other vendor-dependent registers may be provided for messages that are directed to individual units.

2.2. Objectives. The design of the CSRs was intended to meet the following objectives:

- (1) *Scalable.* The architecture should be applicable to a wide range of systems, from the low-cost Serial Bus to the high-performance Futurebus+ and SCI interconnects.
- (2) *Extendable.* The architecture should support existing processor and bus components, while not constraining future system designs, in the following ways:
 - (a) *Addressing.* Many existing processors have 32-bit addresses; these should be sufficient to access resources defined by the CSR Architecture. However, the CSR Architecture should support a natural migration to 64-bit multiprocessor architectures in the future.
 - (b) *I/O Buses.* The architecture should support bridges from standard buses to other I/O buses (such as VME) or vendor-dependent processor/memory buses.
 - (c) The architecture should support bridges between standard buses, such as Futurebus+ or SCI (which could be used as processor/memory buses) and Serial Bus (which is intended for use as an I/O bus).
- (3) *Interoperable.* The architecture should support system configurations with modules supplied by different vendors. The functionality of one vendor's module should not be adversely affected by modules supplied by other vendors.
- (4) *Supportable.* The architecture should be fully supportable. This affects the following design areas:
 - (a) *Self-test.* The node's self-test capabilities, which are invoked before system software is available, should be standardized.
 - (b) *Power Fail Recovery.* The control register definitions should support the recovery by operating system software after a power failure.
 - (c) *Error Logging.* The control register definitions should support the logging of standard, bus dependent, and vendor-dependent error conditions.

- (d) *Fault-retry*. The control register definitions should support the software-initiated retry of transmission-related bus-transaction errors.
- (e) *Autoconfiguration*. The firmware, initialization software, or run-time operating system software need not be changed to configure a new node, bus, or system.
- (f) *Live Insertion*. The CSR Architecture should support the on-line insertion and removal of modules, without requiring a system re-boot, when the bus standard provides that capability.
- (5) *Predictable*. With the appropriate implementation level support, predictable real-time task scheduling should be efficiently supported. Real time support should include the following:
 - (a) *Synchronized Clocks*. The CSR Architecture should provide a mechanism for synchronizing distributed clocks, so each node can maintain a consistent version of the global system time.
 - (b) *Unit-Architecture Options*. The CSR Architecture should provide the opportunity for the definition of prioritized scheduling in the vendor-supplied unit architectures (for example, processors and DMA controllers).
- (6) *Parallelizable*. Multiprocessor configurations require support in the following areas:
 - (a) *Interrupts*. An interrupt architecture should efficiently support interrupts generated by simple I/O nodes, DMA controllers, and other processors.
 - (b) *Coherence*. The transaction sets and DMA architecture should be compatible with the Futurebus+ (broadcast and eavesdrop based) and SCI (distributed directory) cache-coherence protocols.
 - (c) *Message Passing*. A message-passing architecture should efficiently support reception of messages generated by general-purpose processors or special-purpose processors on I/O nodes or DMA controllers.
- (7) *Adaptable*. The I/O driver software should depend on the function that is supported, not the packaging which is selected. If properly implemented, the driver designed to support one unit architecture could be used by either a unit-per-board design or a multiple-unit-per-board design.

Initialization standards are also necessary to guarantee interoperability between nodes supplied by different vendors (without modification of the system-boot firmware). However, these standards are harder to define and affect fewer nodes. For those reasons, the objectives of the CSR Architecture did not include the following initialization standards:

- (1) *Monarch Selection*. A processor (called the monarch) can be selected to perform the system initialization process.
- (2) *Memory Controller*. A standard memory controller can be initialized by firmware, before system software is available.
- (3) *Boot Device*. A standard boot device can be accessed by firmware, before system software is available.

The ROM defined by the CSR Architecture provides the framework for identifying these initialization-related unit architectures as their definitions are refined by the CSR Architecture or bus-dependent standards in the future. The unit-architecture framework can also be used to define vendor-dependent unit architectures, such as processors, frame buffers, disk controllers, bus bridges, and A/D converters.

3. Transaction Set Requirements

3.1 Transaction Overview. Bus transactions are used to communicate between a requester (which initiates the transaction) and a responder (which completes the transaction). For example, a processor is usually a requester and a memory controller is usually a responder. The CSR Architecture defines the transaction set, which shall be provided by a bus standard, and the subset of these, which shall be supported by a memory-controller unit.

A noncoherent transaction consists of two components, called request and response subactions. The request subaction transfers the address, command, and optionally data from the requester to the responder. The response subaction returns the status and optional data from the responder to the requester. For the read and write transactions, data is only transferred in the response and request subactions respectively. Coherent transactions may have bus-dependent components, and are beyond the scope of the CSR Architecture.

A bus may support unified and/or split transactions. In a unified transaction, the request and response subactions are completed indivisibly; other subactions are inhibited between the initiation of the request subaction and the completion of the associated response subaction. In a split transaction, other transactions or subactions may be completed between the request and response subaction phases of one transaction. Split transactions should be supported when the transaction may pass through one or more bridges.

For compatibility between node and unit architectures, the CSR Architecture specifies several properties of bus transactions: their address formats (which are described in Section 4), the transaction commands, the data-transfer sizes, and the returned status codes.

3.2 Read and Write Transactions. Table 3-1 specifies the read and write transactions, which shall be provided by all bus standards and shall be supported by all memory controllers. These are aligned transactions; the address shall be an integer multiple of the transaction size.

Table 3-1 Required Transaction Types	
Transaction	Size (in bytes)
read1	1
read2	2
read4	4
read8	8
read16	16
read64	64
write1	1
write2	2
write4	4
write8	8
write16	16
write64	64

Note that not all of these transactions need be supported by all nodes or all portions of the node address space (simple nodes need only support read4 and write4 transactions). However, the common memory transaction set provides a common expectation for the design of processor and DMA unit architectures.

Only directed **read4** and **write4** transactions are required on all nodes, since these are used to access the CSRs that control the node's operation. On buses that support broadcast write4 transactions, these transactions shall be equivalent to a set of write4 transactions simultaneously directed to all nodes on the local bus.

The read1, read2, read4, read8, read16, write1, write2, write4, write8, write16, lock4, and lock8 transactions are **transported and processed indivisibly**. For example, after two write16 transactions to the same address have completed, the contents of that address shall be equal to the data contained in one of the two write16 transactions (and shall not be a mixed copy of the two).

The read64 and write64 transactions shall be transported indivisibly on standard buses, but the indivisibility of their processing by a responder is unit-architecture dependent. For example, a write64 transaction directed to a message-passing address is processed indivisibly but the indivisibility of a write64 transaction directed to a memory-controller RAM address is not guaranteed.

Lock transactions, which implement indivisible updates on a specified address, shall be provided by CSR Architecture compliant bus standards. The noncoherent lock4 and lock8 transactions may be used to perform indivisible updates at the responder (fetch&add, for example). Although lock transactions encodings shall be provided by a bus standard, their implementation on nodes is optional. These optional transactions are specified in Table 3-2.

Table 3-2 Optional Transaction Types	
Transaction	Size (in bytes)
lock4	4
lock8	8

A bus standard may provide other bus-dependent read and write transactions to improve the efficiency of unaligned or large-block transfers. Unaligned transfers may be more efficiently supported with selected-byte write transactions, which selectively enable updates of bytes within a small aligned data block. Aligned transfers may be efficiently supported by larger block-aligned read or write data transfers (Serial Bus, Futurebus+, and SCI all support an optional 256-byte transfer using a 256-byte aligned address).

A bus standard may also define **move** transactions (also called unacknowledged writes) to improve the efficiency of large-block transfers. A move transaction is acknowledged when accepted by an intermediate agent, but no response is provided when processed by the responder. Transaction ordering constraints (which delay writes until moves have completed) or higher-level protocols are expected to confirm the completion of move transactions, in a bus-dependent fashion.

A bus standard may define other bus-specific transactions (such as initialization-related transactions, cache-coherent accesses, synchronization primitives, or clock synchronization strobes), but these transaction definitions are beyond the scope of the CSR Architecture.

3.3 Noncoherent Lock Transactions. Because of the distributed nature of multiple-bus configurations, the interconnect cannot reasonably be locked while transaction sequences implement indivisible (e.g., test&set) operations on a memory location. Therefore, special atomic transactions called locks are defined. Locks are noncoherent transactions that communicate the intent from the requester to the responder, thus allowing indivisible updates to be performed at the responder.

There are two standard lock transaction formats (4-byte and 8-byte transactions), with several subcommands to define conditional and unconditional update actions that can be used for noncoherent memory accesses. The lock subcommand codes are standardized, to minimize conversion costs when transactions pass through bridges. Buses that implement lock transactions are required to transport the first 8, so they can be used as intermediate buses between other bus standards. The required implementation of these lock transactions on responding nodes (e.g., memory or specialized control registers) is bus-dependent and beyond the scope of the CSR Architecture.

The lock subcommands are based on the model required for implementing the fetch&add and compare&swap primitives. The other subcommands define other update actions that can be easily performed with minimal additions to the basic lock-implementation hardware.

In the simplified lock implementation model two data values (*data* and *arg*) are sent in the lock request; one data value (*old*) is returned in the lock response. These are illustrated in Fig 3-1.

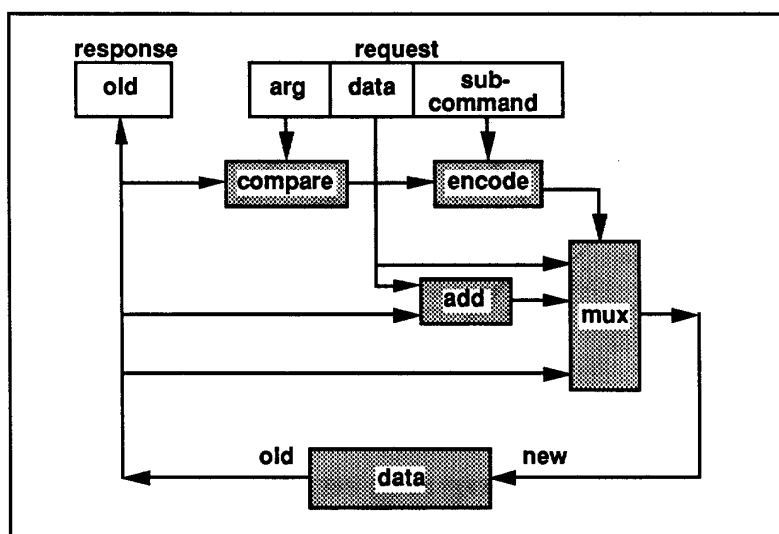


Fig 3-1
Simplified Lock Model

The three data values (*data*, *arg*, and *old*) are all the same size and are either 4 or 8 bytes in size. The subcommand values are standardized, but the backplane signals or fields within packets that are used to transmit these values are bus-dependent. The lock's subcommand field selects which indivisible operation shall be performed at the responder, as specified in Table 3-3. The value of *new* is put in the addressed location when the lock update is performed, based on the lock-transaction subcommand and the values of *data* and *arg* in the request subaction.

Table 3-3
Subcommand Values for Lock4 and Lock8

Code	Name	Update*
0	----	(not used)
1	mask_swap	new= (data&arg) (old&~arg);
2	compare_swap	if (old==arg) new=data; else new= old;
3	fetch_add	(big)new= (big)old+(big)data;
4	little_add	(little)new=(little)old+(little)data;
5	bounded_add	if (old!=arg) new= data+old; else new= old;
6	wrap_add	if (old!=arg) new= data+old; else new= data;
7	vendor-depend	new= op(old,data,arg);
8-15	reserved[8] [†]	new= op(old,data,arg);

* C-code notation used to define update actions.

[†] Reserved encodings for future definition by the CSR Architecture.

Note that the arg value is not used by the fetch_add and little_add subcommands, and need not be transferred to the responder. For these subcommands, a bus standard may specify that the arg value is not transferred (a more efficient transaction format) or may specify that the arg value is transferred (a more uniform transaction format).

Note that the simple swap subcommand is not explicitly defined. However, the mask_swap subcommand (with a mask field of all 1's) can provide the equivalent functionality.

Both big- and little-endian fetch&add lock subcommands are defined (fetch_add and little_add). For the big-endian and little-endian adds, the byte with the smallest address within the addressed integer is assumed to be the most or least significant, respectively. Only big-endian unequal_add and wrap_add lock subcommands, which are expected to be used less frequently, are defined.

3.4 Transaction Errors. Bus standards are expected to provide bus transactions with a status code, to confirm the success of the transaction or localize the cause of the error. The bus standard shall minimally support the following error conditions, which are referenced by the CSR Architecture standard and defined in 1.4:

- (1) **type_error**
- (2) **address_error**
- (3) **conflict_error**
- (4) **response_timeout**

A bus standard may define other types of errors (such as address and parity errors), and may provide additional status bits to specify which node detected the error (e.g., agent or responder) or to localize the error location (address parity, etc.). The details of these error status codes are beyond the scope of this standard. The **done_correct** status is assumed if none of the standard or bus-dependent error status codes are returned.

The requester unit is expected to be informed of errors that affect a transaction it has initiated. This is necessary to support a consistent fault management and error logging strategy and to avoid propagation of corrupt data in the system. The requester's error logging, containment, and recovery strategies are unit-architecture dependent.

An error that is detected in a responder (or agent between the requester and responder) is expected set an error-log bit (STATE_CLEAR.*elog*) after updating the node's ERROR_LOG registers. The optionality of the ERROR_LOG registers and their detailed formats are bus-dependent.

Note that the error status cannot always be returned to the requester, for the following reasons:

- (1) Some bus standards define move transaction (a responseless write transaction);
- (2) Bridges may buffer write transactions (an optimistic done_correct response is returned before the completion status is available); and
- (3) Subactions containing the requester's address can be corrupted.

The logging and containment strategies for such errors (which cannot be reliably reported to the requester) are bus-dependent.

Transaction status codes should be used to report transmission failures. Other types of I/O device status (such as a disk off-line status or correctable-RAM error) are expected to be reported through software-accessible unit-dependent control registers.

On a broadcast transaction, the standard bus may ignore errors (broadcasts are not required to be reliable) or may provide mechanisms for summarizing multiple error-status reports. Unrecoverable errors (type_error, address_error, and bus-dependent transmission errors) should have precedence over conflict_error transaction-status reports.

3.5 Immediate Effects. When accessing a CSR defined by the CSR Architecture, the transaction's effects shall always be immediate, in that they appear to occur after the acceptance of the request and before the generation of the response. If initialization tests are supported, for example, a write to the node's RESET_START register has the effects of immediately moving the node to the initializing state and starting an initialization test. Both of these effects are immediate, although the completion of the initialization test may be delayed.

Writes to several of the CSRs have externally visible effects, which can immediately affect the acceptance as well as processing of other transactions. For example, a write to the RESET_START register immediately disables the extended units and extended memory spaces. The writes that affect the acceptance of other transactions are not expected to be buffered (completed in FIFO order after the response has been returned), since their effects shall be immediately visible.

For split transactions, several request subactions may be accepted before the first one is processed and several requests may be processed before the first response has been returned. A request subaction is accepted, the request may be temporarily queued before being processed, the request processing produces a response subaction, the response may be temporarily queued, and the response is eventually sent, as illustrated by the split-response processing model of Fig 3-2.

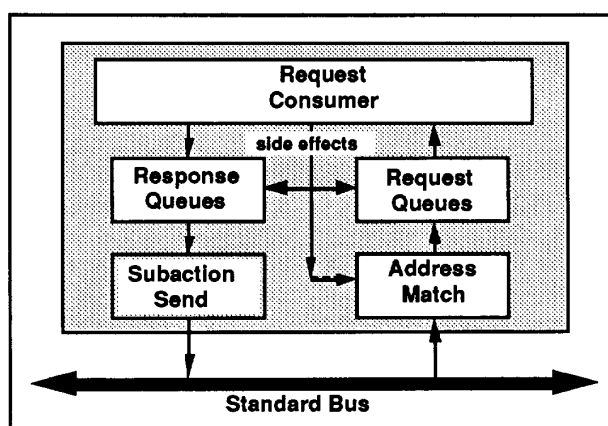


Fig 3-2
Responder CSR Processing Model

With split-response transactions, all transaction-processing effects shall occur simultaneously after the request subaction has been removed from the request queue (the generation of a response subaction is one of these effects). The request and response queues enqueue/dequeue order is not constrained by the CSR Architecture (a FIFO queue model is not required).

The effects of some CSR writes may delete previously queued request or response subactions; for example, a write to the node's RESET_START register may discard queued request and response queue entries. Active responses, which are being retransmitted after having been previously busied, may also be discarded.

The effects of CSR writes may also affect which requests are accepted based on their address; for example, a write to the node's MEMORY_BASE register changes the address-match parameters and affects which request subactions are accepted in the future. Changes of the node's address-match parameters may have bus-dependent effects on the processing of previously queued requests (which were accepted based on the old address).

To avoid bus-dependent effects, responder queues are expected to be empty when software writes to a CSR that affects the address-match parameters or deletes request-queue or response-queues entries. These CSR locations are summarized in Table 3-4.

Table 3-4
CSRs With Special Split-Responder Effects

CSR Location	Special Considerations
STATE_CLEAR	The <i>off</i> bit discards queue entries, <i>lost</i> bit disables address spaces.
RESET_START	May discard request-queue and response-queue entries.
NODE_IDS	Changes address-match (all addressing models).
UNITS_BASE	Changes address-match (32/64-bit extended addressing).
UNITS_BOUND	"
MEMORY_BASE	"
MEMORY_BOUND	"

Note that a write to the NODE_IDS register changes the responder's node_id parameter, which (on some buses) is included as a source_id parameter in the response subaction. These bus standards should clearly define the meaning of this source_id field (i.e., is it the old or new node_id value?).

4. Node Addressing

4.1 Node Addresses. The CSR Architecture defines two largely compatible address-space mappings, as follows:

- (1) *Extended.* The extended address-space model supports 32-bit and 64-bit addresses. The 32-bit extended address space is split into available and register address space. The register address space is partitioned into 64K initial node spaces. The available space can be dynamically assigned to increase the size of address spaces mapped to a node. A bus standard may support only 32-bit addresses or 32-bit and 64-bit addresses.
- (2) *Fixed.* The fixed address-space model supports only 64-bit addresses. The entire 64-bit address space is split into 64K equal initial node spaces, which are 256 Tbytes in size. A node space contains initial memory, private, register, and initial units spaces.

Since some of the bus addresses and node offset addresses are reserved for special uses, this restricts the total number of realizable buses and nodes to a slightly smaller number.

The same number of node addresses (64K) are supported in both address-space mappings, to simplify the address conversions through bus bridges. See the following sections for details.

4.2 Extended Addressing. The extended addressing is based on 32-bit and 64-bit physical address spaces, which are both split into separate available and register address spaces. The register address space is partitioned into bus offset addresses, and each bus is partitioned into node offset addresses. Although a large number of nodes are mapped to the register address space, the register address space is only a small portion of the total physical address space. The 32-bit extended address space layout is illustrated in Fig 4-1.

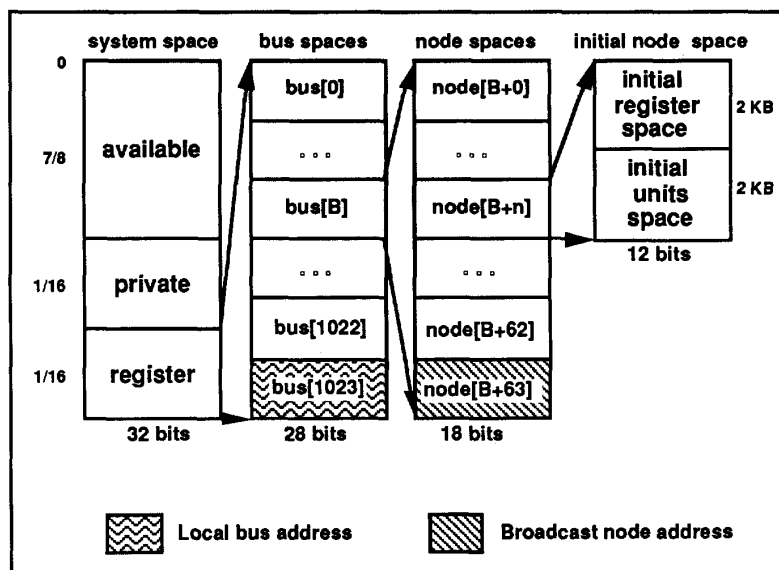


Fig 4-1
32-Bit Extended Addresses

The 32-bit extended address model supports the available, private, and register spaces, whose address ranges are specified in Table 4-1.

Table 4-1
32-Bit Extended Addresses

address_base	address_bound	Description
00000000	DFFFFFFF	32-bit available space
E0000000	FFFFFFF	32-bit private space
F0000000	FFFFFFF	Register space

Within the register space, up to 64 node addresses are allocated to one bus, and up to 1024 bus addresses are allocated to each system. Since some of these allocated addresses have special uses, this limits the system to 1023 buses with 63 nodes on each bus.

The node's initial address space is limited to 4 Kbytes, which contains two address spaces: the node's initial register space and initial units space. The address of the node is specified by a 16-bit *node_id* value; the 10 MSBs (most-significant bits) define the *bus_id* and the 6 LSBs (least-significant bits) define the *offset_id*. Software is expected to configure the *node_id* values so that all nodes on one bus have the same *bus_id* value and a bus-unique *offset_id* value.

One of the *bus_id* values (1023) has a special meaning: all slaves respond to this fixed local bus address as well as to their dynamically configured global bus address. Software is expected to use the fixed local bus address to access nodes on the local bus before the global bus address has been assigned. After the global bus addresses have been assigned, software is expected to use global bus addresses to access all nodes.

If broadcast transactions are supported by the bus standard, one of the *offset_id* addresses (63) has a special meaning; all slaves respond to this *offset_id* address as well as their locally distinct *offset_id* number. Software is expected to use the broadcast *offset_id* address to broadcast write transactions to the nodes attached to one bus. A write to *bus_id* 1023 and *offset_id* 63 is broadcast to all nodes on the local bus (including the requester). A write to a remote *bus_id* address with an *offset_id* of 63 is broadcast to all nodes on a remote bus. If broadcast transactions are not supported by the bus standard, this *offset_id* address (63) is not used.

Broadcast read transactions (sometimes called broadcast transactions) are not supported by the CSR Architecture. Read or lock transactions directed to a broadcast node address (node 63) have unspecified side effects and may compromise system integrity. To improve its error-detection capabilities, a node may detect the illegal read or lock address before the transaction is initiated on the bus.

The **64-bit extended** address space is an extension of the 32-bit extended address space. The lowest and highest portions of the 64-bit extended addresses correspond to 32-bit addresses; the middle of the 64-bit address space has no 32-bit extended address equivalent. The mapping between the 32-bit extended addresses and portions of the 64-bit extended address space is illustrated in Fig 4-2.

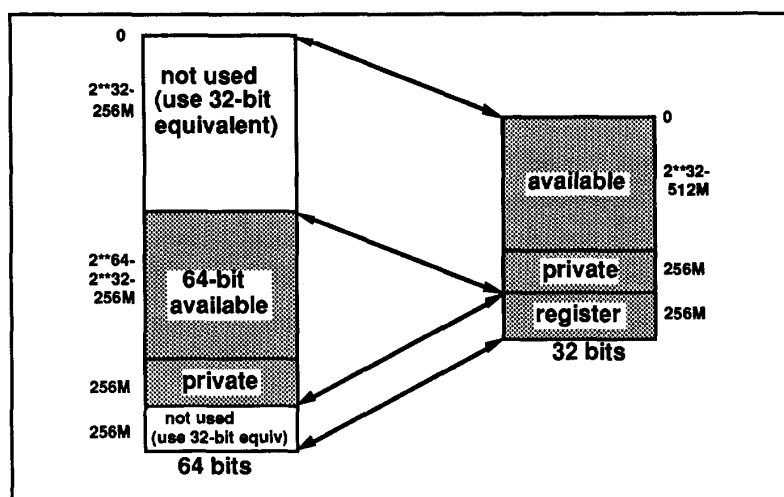


Fig 4-2
64-Bit Extended Addresses

On buses using the 64-bit extended address-space model, all nodes shall have the capability of responding to 32-bit addresses and may optionally have the capability of responding to 64-bit addresses as well. To ensure compatibility between 32-bit-only and 32/64-bit nodes, the 64-bit masters shall pre-decode their 64-bit addresses before they are asserted on the bus. If the 64-bit address has a 32-bit equivalent, the 32-bit address equivalent (the least-significant portion of the 64-bit address) shall be used, as described in Table 4-2. This address translation is expected to be performed by bus-interface logic, but may also be performed by virtual-to-physical translation tables in a processor.

Table 4-2
64-Bit Extended Addresses

address_base	address_bound	Description
00000000 00000000	00000000 EFFFFFFF	Prohibited (use 32-bit available space)
00000000 F0000000	FFFFFFFF DFFFFFFF	64-bit available space
FFFFFFFF E0000000	FFFFFFFF EFFFFFFF	64-bit private space
FFFFFFFF F0000000	FFFFFFFF FFFFFFFF	Prohibited (use 32-bit register space)

Because the entire 32-bit address space is contained within a subset of the 64-bit address space, a 64-bit physical address pointer is sufficient to specify any address on a hybrid 32/64 bit system (a special bit is not required to specify which address space is used).

4.3 64-Bit Fixed Addressing. The 64-bit fixed addressing partitions the address into separate 16-bit node_id and 48-bit address_offset components, to simplify routing through active switches. The 64-bit address is partitioned into 64K equal-sized initial node spaces (which need not be associated with specific bus addresses). Although the address space is

shared by a large number of nodes, the address space assigned to each node is still large (256 Tbytes), as illustrated in Fig 4-3.

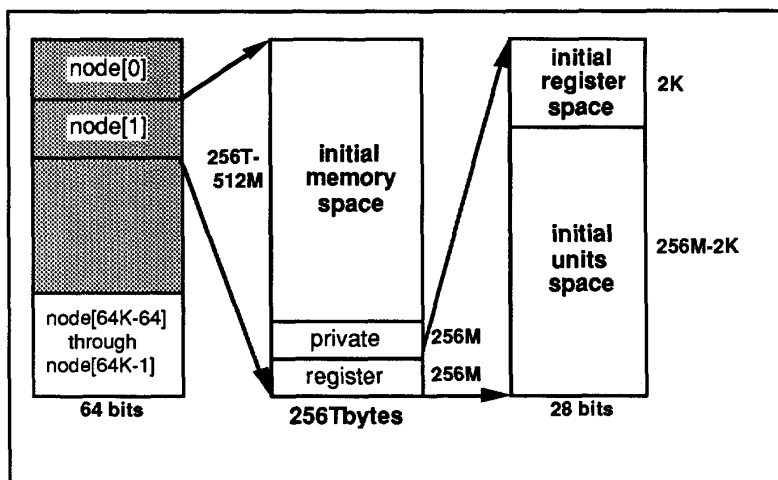


Fig 4-3
64-Bit Fixed Addressing

Up to 64 of the largest node_id addresses are allocated for bus-dependent uses. The shaded blocks illustrates the addresses that are available for nodes.

Since the address space of each node (48 bits) is sufficient to meet any foreseen requirements, address extension registers shall not be implemented. By binding the memory addresses to the node's initial address space, the cost and complexity of address recognition hardware is reduced.

When practical, the node should use only the first 2 Kbytes of the initial units space. This simplifies the design of heterogeneous bus systems, since the initial register and 2 Kbytes of the initial units spaces can be directly mapped to their 32-bit equivalents on buses using extended addressing.

When implemented, a memory controller is expected to be mapped to a contiguous range of the node's initial memory space, beginning at zero. If the initial units space is insufficient, additional units may also be placed within the initial memory space.

4.4 Private Addresses. The private address space is allocated for vendor-dependent node-local uses. Some nodes may need to allocate the private space for internal uses, in a way that makes the private space on the bus inaccessible to these nodes. Such nodes, which are said to implement the private space, shall not generate bus transactions with addresses in the private space. Instead, transactions with private space addresses should be routed to node-local resources (such as a boot ROM).

Nodes that do not implement the private space can access these addresses on the bus. However, software is expected to constrain the assignment of extended spaces within the private space, since these bus addresses cannot be accessed by those nodes that implement the private space option.

4.5 Initial Node Space. A node initially responds to a 4-Kbyte initial node space. Half of the initial node space (2 Kbytes) is register space and the other half is allocated to unit-dependent specifications. One quarter of the 2-Kbyte register space is allocated to CSRs (as

defined by this document), one quarter of the register space is allocated to bus-dependent registers (which are restricted by this document), and the other half provides an address window to the initial addresses in the node's ROM, as illustrated in Fig 4-4.

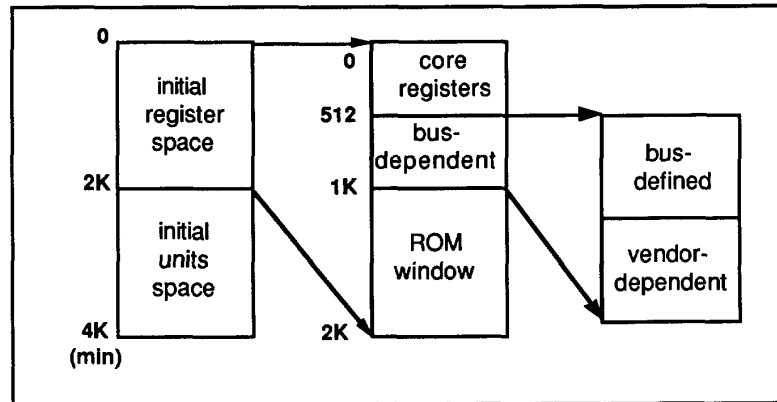


Fig 4-4
Initial Node Space Components

Read transactions to the initial register space shall be supported and shall have no side effects. This restriction is intended to simplify state-save, diagnostics, and fault-retry software, which can safely read (or reread) any address within the initial register space (without unexpected, and sometimes unrecoverable, side effects).

The bus standard shall define the first part of the bus-dependent portion of the initial node space, and shall allocate the remainder for vendor-dependent uses. The bus standard shall specify the number of vendor-dependent registers at the end of the bus-dependent portion of the initial node space.

4.6 Extended Address Spaces. In the extended address space model, the initial size of a node is 4 Kbytes, which may be insufficient for many applications (for examples, a memory controller or memory-mapped graphics). To increase the effective size of the node address space, two pairs of address-extension registers are provided; these allow the node to additionally respond to an extended units space and an extended memory space.

A node's extended units or extended memory spaces are expected to be assigned in the **available address space**. The available address space is initially empty, and (since the node's extension addresses can be dynamically assigned to this space by software) the CSR Architecture places no restrictions on the use of the available space. However, software is expected to assign extended memory addresses to a contiguous range of addresses starting from address zero.

A multifunction node may contain a memory controller as well as other I/O unit architectures (such as a graphics frame buffer). By providing separate registers for extending the size of the memory and units spaces, the extended memory spaces and the extended units spaces can be assigned independently. Thus, the extended memory spaces of several memory controllers can be assigned contiguously, as illustrated (for two extended memory spaces) in Fig 4-5.

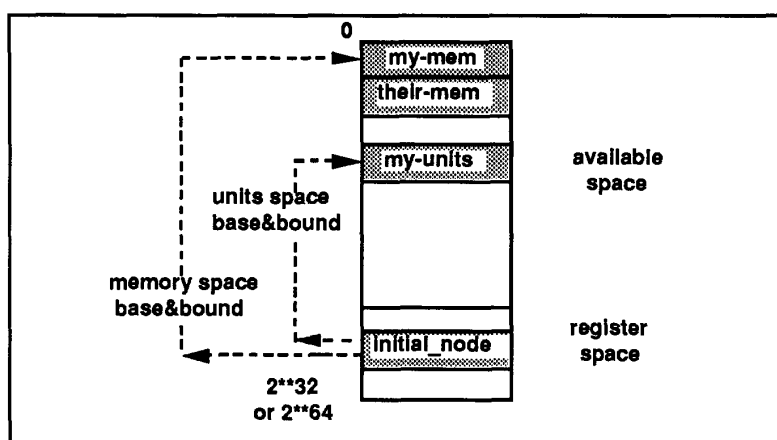


Fig 4-5
Configurable Extended Addresses

Note that (for each node) only one extended units space is required, since multiple units (such as memory-mapped graphics and processors) can be mapped into separate partitions in the extended units space.

The location and size of an extended space is specified by two registers; the base register specifies the lowest address in the extended space and the bound register specifies the next highest address (one more than the last address in the extended space). The base register contains an enable bit, which can be used to selectively disable an extended space while it is being moved.

To simplify hardware address decoders, alignment restrictions are placed on the location of the extended address spaces. For example, a node that supports an 8 Mbyte memory controller may restrict the base location of its extended memory space to addresses that are a multiple of 1 Mbyte in size. The alignment and size restrictions for the extended memory and extended units space are specified by ROM-supplied values; see 8.4.13 and 8.4.14 for details.

Except for the address alignment and size restrictions, a node shall place no restrictions on the location of its extended spaces—a node may have its extended space located in any, or across several, of the available, private, and register spaces.

4.7 Indirect Space. A node shall provide an indirect space and a standardized ROM shall be located at the lower indirect space addresses. The first 1 Kbyte of indirect address space is directly mapped to a 1 Kbyte portion of the node's 4-Kbyte initial node address space. This simplifies inexpensive nodes, of which the total ROM size is less than 1 Kbyte in size. When larger ROMs are implemented, critical information, such as the type of node and bus-dependent parameters, shall be located at the beginning of ROM. This directly mapped ROM address space is illustrated in Fig 4-6.

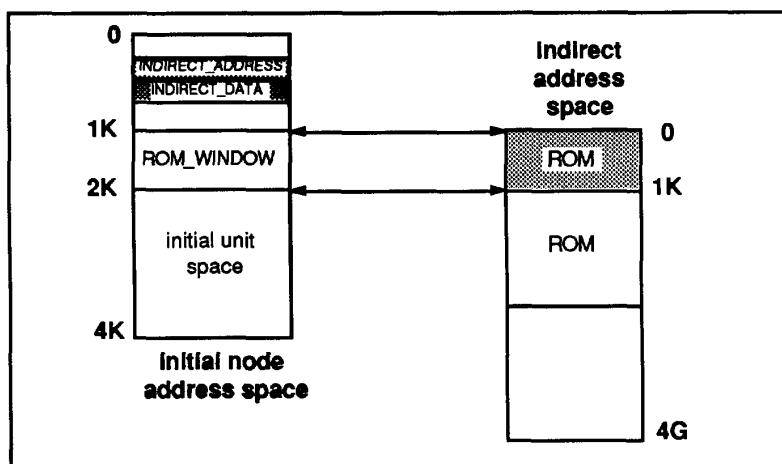


Fig 4-6
Indirect Space Mapping (address less than 1 Kbyte)

In addition to the ROM, the indirect space may contain unit-dependent registers, which may have read and write capabilities. When accessing these registers, reads shall have no side effects.

Other indirect addresses, those with addresses at or above 1024 in the indirect space, shall be indirectly accessed through the **INDIRECT_ADDRESS** and **INDIRECT_DATA** register pair. When accessed indirectly, a write to the **INDIRECT_ADDRESS** register establishes the indirect address; a following read of the **INDIRECT_DATA** register returns a quadlet from this pre-specified indirect address. In a multiprocessor environment, higher level locks are expected to ensure that this sequence is performed indivisibly. This indirect access model (which is used on indirect addresses above 1 Kbyte) is illustrated in Fig 4-7.

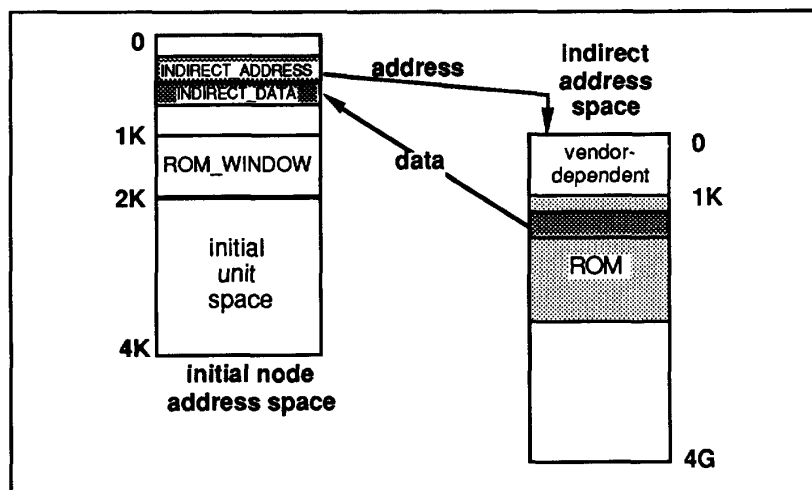


Fig 4-7
Indirect Space Mapping (address greater or equal to 1 Kbytes)

Software is always expected to access the first 1 Kbyte of the indirect address space by accessing an address within the node ROM_WINDOW. When the value of the INDIRECT_ADDRESS is less than 1 Kbyte, the value returned from a read of the INDIRECT_DATA register and the effect of a write to the INDIRECT_DATA register are both vendor-dependent.

4.8 Address Space Offsets. The node's ROM contains entries that specify the location of unit-related resources within the initial units space and within the extended units space. When the specified address is contained within the initial units space, the address of the resource is measured as a byte offset from the beginning of the initial node space, as illustrated in Fig 4-8.

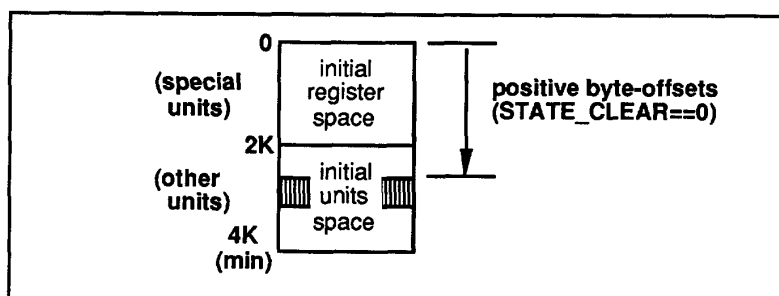


Fig 4-8
ROM-Specified Offsets in the Initial Units Space

When the specified resource is contained within the extended units or initial memory spaces, the address of the resource is measured as a byte-offset from the beginning of the space, as illustrated in Fig 4-9.

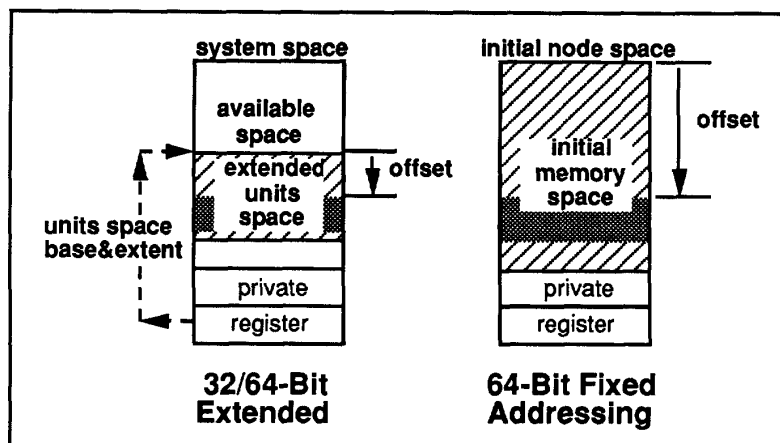


Fig 4-9
ROM-Specified Offsets in Extended Units Space