

**Implementation of
ISO/IEC 13213 : 1994**

**Information technology —
Microprocessor systems —
Control and Status Registers
(CSR) Architecture for
microcomputer buses**

Committees responsible for this British Standard

The preparation of this British Standard was entrusted by Technical Committee IST/6, Data communications, to Technical Subcommittee IST/6/10, IEC/CLC monitoring subcommittee, upon which the following bodies were represented:

British Telecommunications plc
Digital Equipment Co. Ltd.
Electrical Contractors' Association
Electricity Association
Institution of Electrical Engineers
Nine Tiles Computer Systems Ltd.

This British Standard, having been prepared under the direction of the Information Systems Technology Assembly, was published under the authority of the Standards Board and comes into effect on 15 September 1995

© BSI 1995

The following BSI references relate to the work on this standard:
Committee reference IST/6/10
Draft for comment 92/68183 DC

ISBN 0 580 24379 6

Amendments issued since publication

Amd. No.	Date	Text affected

National foreword

This British Standard reproduces verbatim ISO/IEC 13213 : 1994 and implements it as the UK national standard.

This British Standard is published under the direction of the Information Systems Technology Assembly whose Technical Subcommittee IST/6/10 has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international committee any enquiries on interpretation, or proposals for change, and keep UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

NOTE. International and European Standards, as well as overseas standards, are available from Customer Services, BSI, 389 Chiswick High Road, London W4 4AL.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

INTERNATIONAL
STANDARD

ISO/IEC
13213

ANSI/IEEE
Std 1212

First edition
1994-10-05

**Information technology –
Microprocessor systems – Control
and Status Registers (CSR) Architecture
for microcomputer buses**

*Technologies de l'information –
Systèmes à microprocesseurs – Architecture
des registres de commande et d'état
pour bus de micro-ordinateur*



Reference number
ISO/IEC 13213: 1994(E)
ANSI/IEEE
Std 1212, 1994 Edition

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

ISBN 1-55937-448-9

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

October 5, 1994

SH94220

ISO/IEC 13213 : 1994
[ANSI/IEEE Std 1212, 1994 Edition]
(Incorporates ANSI/IEEE Std 1212-1991)

Information technology— Microprocessor systems—Control and Status Registers (CSR) Architecture for microcomputer buses

Sponsor

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



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



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



Foreword

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

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

In 1994, ANSI/IEEE Std 1212-1991 was adopted by ISO/IEC JTC1, as draft International Standard ISO/IEC DIS 13213. This edition incorporates editorial comments received in the review of ISO/IEC DIS 13213.



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

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

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

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

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

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

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

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

Introduction

(This introduction is not a part of this International Standard or of ANSI/IEEE Std 1212, 1994 Edition.)

Bus standards have often been set 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.

The following is a list of participants in the IEEE Control and Status Register (CSR) Working Group at the time ANSI/IEEE Std 1212-1991 was approved:

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 Mentzelopoulos
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
Michael 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 Kirmann
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

IEEE Std 1212-1991 was approved by the American National Standards Institute on May 14, 1992.

Contents

CLAUSE	PAGE
1. Document structure and notation	1
1.1 Document structure	1
1.2 References	1
1.3 Conformance levels	1
1.4 Technical glossary	2
1.5 Bit, byte, and quadlet ordering	9
1.6 Numerical values	9
1.7 C code notation	10
1.8 CSR, ROM, and field notation	10
1.9 Register specification format	11
1.10 Reserved registers and fields	12
2. Objectives and scope	15
2.1 Scope	15
2.2 Objectives	15
3. Transaction set requirements	17
3.1 Transaction overview	17
3.2 Read and write transactions	17
3.3 Noncoherent lock transactions	18
3.4 Transaction errors	20
3.5 Immediate effects	21
4. Node addressing	23
4.1 Node addresses	23
4.2 Extended addressing	23
4.3 64-bit fixed addressing	25
4.4 Private addresses	26
4.5 Initial node space	26
4.6 Extended address spaces	27
4.7 Indirect space	28
4.8 Address space offsets	29
5. Node architectures	31
5.1 Modules, nodes, and units	31
5.2 Node states	32
5.3 Node testing	33
5.3.1 Access-path tests	33
5.3.2 Reset test	33
5.3.3 Diagnostic tests	34
5.3.4 Non-standard diagnostic tests	35
5.4 Multinode modules	36
5.5 On-line replacement (OLR)	36
6. Unit architectures	39

CLAUSE	PAGE
6.1 Unit architecture overview	39
6.2 Interrupts	39
6.2.1 Interrupt-target registers	39
6.2.2 Interrupt-poll registers	40
6.3 Message passing	41
6.4 Globally synchronized clocks	41
6.4.1 Clock overview	41
6.4.2 Clock synchronization	42
6.4.3 Clock update models	43
6.4.4 Updating clock registers	44
6.4.5 Clock accuracy requirements	45
6.5 Memory unit architectures	45
6.6 Unit architecture environment	45
7. CSR definitions	47
7.1 Register names and offsets	47
7.2 Minimal implementations	50
7.3 Unsupported register accesses	51
7.4 Register definitions	51
7.4.1 STATE_CLEAR	51
7.4.2 STATE_SET	54
7.4.3 NODE_IDS	55
7.4.4 RESET_START	56
7.4.5 INDIRECT_ADDRESS	57
7.4.6 INDIRECT_DATA	58
7.4.7 SPLIT_TIMEOUT	58
7.4.8 ARGUMENT	59
7.4.9 TEST_START	61
7.4.10 TEST_STATUS	64
7.4.11 UNITS_BASE	66
7.4.12 UNITS_BOUND	68
7.4.13 MEMORY_BASE	69
7.4.14 MEMORY_BOUND	70
7.4.15 INTERRUPT_TARGET	71
7.4.16 INTERRUPT_MASK	72
7.4.17 CLOCK_VALUE	72
7.4.18 CLOCK_TICK_PERIOD	74
7.4.19 CLOCK_STROBE_ARRIVED	75
7.4.20 CLOCK_STROBE_INFO	76
7.4.21 Message targets	76
7.4.22 ERROR_LOG registers	77
8. ROM specification	79
8.1 Introduction	79
8.1.1 ROM design assumptions	79
8.1.2 ROM formats	79
8.1.3 Driver and diagnostic identifiers	79
8.1.4 ASCII text	81
8.1.5 CRC calculations	81
8.2 ROM formats	83

CLAUSE	PAGE
8.2.1 First ROM quadlet	83
8.2.2 Minimal ROM format	83
8.2.3 General ROM format	83
8.2.4 Directory formats	84
8.2.5 Leaf format.....	86
8.2.6 Textual_descriptor	86
8.3 bus_info_block.....	88
8.4 Root directory entries.....	89
8.4.1 Bus_Dependent_Info	90
8.4.2 Module_Vendor_Id.....	90
8.4.3 Module_Hw_Version.....	90
8.4.4 Module_Spec_Id	91
8.4.5 Module_Sw_Version	91
8.4.6 Module_Dependent_Info	91
8.4.7 Node_Vendor_Id.....	91
8.4.8 Node_Hw_Version.....	91
8.4.9 Node_Spec_Id	91
8.4.10 Node_Sw_Version	92
8.4.11 Node_Capabilities	92
8.4.12 Node_Unique_Id.....	92
8.4.13 Node_Units_Extent.....	92
8.4.13.1 Node_Units_Extent immediate format.....	93
8.4.13.2 Node_Units_Extent offset format.....	94
8.4.14 Node_Memory_Extent.....	95
8.4.14.1 Node_Memory_Extent immediate format.....	95
8.4.14.2 Node_Memory_Extent offset format	96
8.4.15 Node_Dependent_Info	96
8.4.16 Unit_Directory	96
8.5 Unit directories.....	96
8.5.1 Unit_Spec_Id	97
8.5.2 Unit_Sw_Version.....	97
8.5.3 Unit_Dependent_Info.....	97
8.5.4 Unit_Location	97
8.5.5 Unit_Poll_Mask	98
8.6 Key definitions.....	98
8.7 company_ids	99
8.7.1 company_id assignments	99
8.7.2 company_id mappings	100
9. Bus standard requirements.....	101
ANNEXES	
A. Bibliography (Informative).....	103
B. Bus topologies (Informative).....	105
B.1 Specialized buses	105
B.1.1 Multiple-bus topologies	105
B.1.2 Dual-port nodes.....	106
B.2 Fault retry protocols.....	106

ANNEXES	PAGE
B.2.1 Hardware fault recovery	106
B.2.2 Software fault recovery	107
C. System initialization (Informative).....	109
C.1 System initialization summary	109
C.2 Node address assignments	109
C.3 Processor-cache model.....	110
C.4 Address protection	110
C.5 Power distribution models	111
D. Bus transactions (Informative)	113
D.1 Transaction overview	113
D.2 Transaction components	113
D.3 Request subaction fields	113
D.4 Response subaction fields	114
E. Bus bridges (Informative).....	117
E.1 Address-invariant mappings	117
E.2 Transaction forwarding.....	118
E.3 Transaction ordering	119
E.3.1 Split-response transaction ordering.....	119
E.3.2 Buffered-write transparency.....	119
E.3.3 Weakly ordered move transactions	120
E.3.4 Queue-dependency deadlocks.....	121
E.4 Address domains.....	122
E.5 Protection boundaries.....	124
E.6 Coherence domains	124

Information technology—Microprocessor systems—Control and Status Registers (CSR) Architecture for microcomputer buses

1. Document structure and notation

1.1 Document structure

This International Standard 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 autoconfigure 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.

The annexes provide background for understanding the usage of this CSR Architecture specification. The CSR Architecture provides the specification upon which conforming designs should be based. The annex clauses illustrate ways that these capabilities could be used. Note that the annexes are nonbinding.

1.2 References

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

ANSI/ISO/IEC 9899:1990, Programming Languages—C.^{1,2}

ISO/IEC 646:1991, Information technology—ISO 7-bit coded character set for information interchange.²

1.3 Conformance levels

Several keywords are used to differentiate among various levels of requirements and optionality, as follows:

1.3.1 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.

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

¹ Replaces ANSI X3.159-1989.

² ISO documents are available from ISO Central Secretariat, 1 rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse; and from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036-8002, USA.

1.3.3 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.

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

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:

1.4.1 active test: An ongoing 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.

1.4.2 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 acknowledgment is not observed within a fixed timeout period.

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

1.4.4 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.

1.4.5 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.

1.4.6 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.

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

1.4.8 board: The physical component that is inserted into one of the backplane connectors.

1.4.9 bridge: A hardware adapter that forwards transactions between buses.

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

1.4.11 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.

1.4.12 bus-dependent: A term used to describe parameters that are defined by the bus standards that conform to this standard. Although the CSR Architecture may constrain the definition of these fields, their detailed definition is provided by the appropriate bus standard.

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

1.4.14 byte: A byte is 8 bits of data.

1.4.15 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.

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

1.4.17 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.

1.4.18 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.

1.4.19 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.

1.4.20 CSR Architecture: A term that refers to this International Standard.

1.4.21 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.

1.4.22 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.

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

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

1.4.25 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.

1.4.26 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.

1.4.27 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.

1.4.28 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.

1.4.29 doublet: Two bytes (16 bits) of data.

1.4.30 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.

1.4.31 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.

1.4.32 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.

1.4.33 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.

1.4.34 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.

1.4.35 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.

1.4.36 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.

1.4.37 fixed addressing (64-bit): An address model implemented by bus standards supporting only 64-bit addresses. The initial node space is large (256 terabytes), is fixed in size, and extended spaces are not supported.

1.4.38 Futurebus+: A name that refers to ISO/IEC 10857:1994 (ANSI/IEEE 896.1, 1994 Edition) and companion 896.x standards [B3]³. Futurebus+ defines a physically based backplane bus standard, which supports 32-bit and 64-bit physical addresses.

1.4.39 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.

1.4.40 gigabyte: 2^{30} bytes.

1.4.41 immediate effect: An effect of a transaction that 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.

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

1.4.43 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.

1.4.44 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.

³ The numbers in brackets refer to those in annex A.

1.4.45 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 size of initial units space is 4 kbytes. On buses implementing 64-bit fixed addressing, the size of the initial node space is 256 terabytes and also includes the node's initial memory and private spaces.

1.4.46 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.

1.4.47 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.

1.4.48 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.

1.4.49 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 that is entered immediately after a power_reset or command_reset event.

1.4.50 kbyte: kilobyte. Indicates 2^{10} bytes.

1.4.51 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.

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

1.4.53 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.

1.4.54 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.

1.4.55 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.

1.4.56 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.

1.4.57 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.

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

1.4.59 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.

1.4.60 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.

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

1.4.62 MULTIBUS II⁴: Refers to IEEE Std 1296-1987, IEEE Standard for a High-Performance Synchronous 32-Bit Bus: MULTIBUS II (ANSI).

1.4.63 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).

1.4.64 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.”

1.4.65 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 addresses 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.

1.4.66 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.

1.4.67 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).

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

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

1.4.70 NuBus⁵: Refers to IEEE Std 1196-1987, IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus (ANSI).

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

1.4.72 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.

1.4.73 primary state: The state on a node that is initialized by a `power_reset`. For example, the control and status registers defined within this standard are part of the node's primary state.

⁴ MULTIBUS II is a registered trademark of Intel Corporation.

⁵ NuBus is a registered trademark of Texas Instruments, Inc.

1.4.74 quadlet: Four bytes (32 bits) of data.

1.4.75 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.

1.4.76 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.

1.4.77 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.

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

1.4.79 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.

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

1.4.81 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.

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

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

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

1.4.85 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 is defined.

1.4.86 SCI (Scalable Coherent Interface): The name that refers to IEEE Std 1596-1992 [B8]. Though functionally behaving as a bus, the SCI's physical implementation is a collection of point-to-point unidirectional links. These links may be connected to a switch (in high-performance systems) or daisy-chained to form a ring (in low-cost systems).

1.4.87 Serial Bus: The name that refers to the IEEE project, P1394 [B7], which specifies a serial bus intended as a low-cost peripheral connect or an alternate diagnostic and control path.

1.4.88 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.

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

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

1.4.91 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).

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

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

1.4.94 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.

1.4.95 terabyte: 2^{40} bytes.

1.4.96 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.

1.4.97 transaction: A transfer between requester and responder consisting of a request and a 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.

1.4.98 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.

1.4.99 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 calls this a connected transaction. A transaction that is not unified is called a split transaction.

1.4.100 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).

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

1.4.102 unit-dependent: A term used to describe parameters that may vary among 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.

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

1.4.104 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 transaction to the 4-byte STATE_CLEAR register is unsupported.

1.4.105 vendor-dependent: A term used to describe parameters that may vary among 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.

1.4.106 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). 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 figure 1.

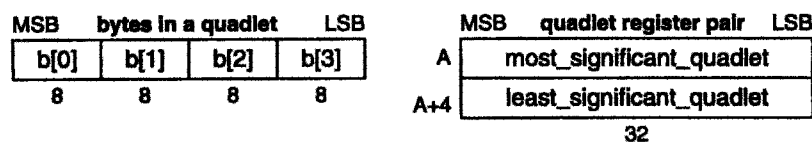


Figure 1—Byte and quadlet ordering

Since 64-bit addressing is supported throughout this standard, many values are stored as quadlet register pairs. For consistency, the quadlet register with the smaller address is also the more significant, as illustrated in figure 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 sizes 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 meanings 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. The detailed definitions of these operations are provided in ANSI/ISO/IEC 9899:1990, Programming Languages—C.

Table 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
$c \ll d$	integer c , shifted left by d bits
$c \gg d$	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, bold type 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 2.

Table 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, as described in 4.2 and 4.3.

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 define accurately 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 figure 2.

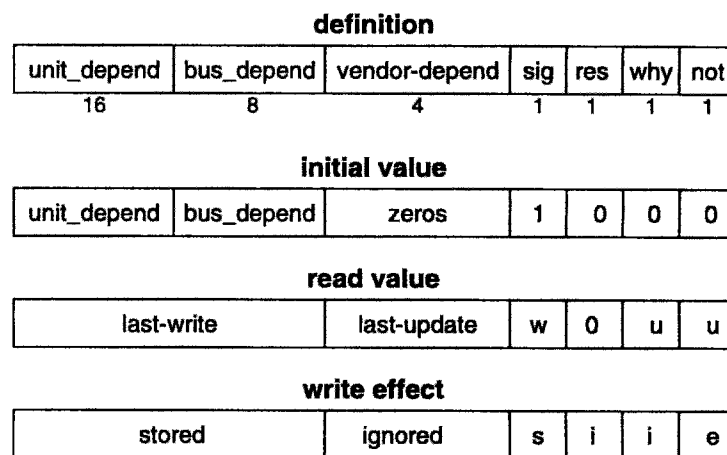


Figure 2—CSR format specification (example)

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 nonvolatile 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	<i>w</i>
Definition	The value of the data field shall be the value that was previously written to the same register address.

last_update:

Abbreviation	<i>u</i>
Definition	The value of the data field shall be the last value that was updated by node hardware.

The following write-effect fields have a generic meaning:

stored:

Abbreviation	<i>s</i>
Definition	The value of the written data field shall be immediately visible to reads of the same register.

ignored:

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

effect:

Abbreviation	<i>e</i>
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 figure 3.

Within an implemented register, a field that is reserved for future revisions of this standard is labeled *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:

definition
unimplemented
32
initial value
zeros
initial&read values
zeros
write effect
ignored

Figure 3—Unimplemented register (hardwired to zero)

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 clause 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:

- a) *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.
- b) *Common Transaction Sets.* A common transaction set (including error-status codes) is defined. This transaction set can be transparently passed through bridges.
- c) *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.
- d) *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.
- e) *Interrupts.* Standard target addresses are provided for interrupts that are broadcast on the bus to all nodes, or broadcast within the node (nodecast) to multiple units. Other vendor-dependent quadlet registers may be provided for interrupts that are directed to individual units.
- f) *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:

- a) *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.
- b) *Extendable.* The architecture should support existing processor and bus components, while not constraining future system designs, in the following ways:
 - 1) *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.
 - 2) *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.
 - 3) 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).
- c) *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.
- d) *Supportable.* The architecture should be fully supportable. This affects the following design areas:
 - 1) *Self-test.* The node's self-test capabilities, which are invoked before system software is available, should be standardized.

- 2) *Power fail recovery.* The control register definitions should support recovery by operating system software after a power failure.
- 3) *Error logging.* The control register definitions should support the logging of standard, bus dependent, and vendor-dependent error conditions.
- 4) *Fault-retry.* The control register definitions should support the software-initiated retry of transmission-related bus-transaction errors.
- 5) *Autoconfiguration.* The firmware, initialization software, or run-time operating system software need not be changed to configure a new node, bus, or system.
- 6) *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.
- e) *Predictable.* With the appropriate implementation-level support, predictable real-time task scheduling should be efficiently supported. Real-time support should include the following:
 - 1) *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.
 - 2) *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).
- f) *Parallelizable.* Multiprocessor configurations require support in the following areas:
 - 1) *Interrupts.* An interrupt architecture should efficiently support interrupts generated by simple I/O nodes, DMA controllers, and other processors.
 - 2) *Coherence.* The transaction sets and DMA architecture should be compatible with the Future-bus+ (broadcast and eavesdrop based) and SCI (distributed directory) cache-coherence protocols.
 - 3) *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.
- g) *Adaptable.* The I/O driver software should depend on the function that is supported, not the packaging that is selected. If properly implemented, the driver designed to support one unit architecture could be used either by a unit-per-board design or by a multiple-unit-per-board design.

Initialization standards are also necessary to guarantee interoperability among 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:

- a) *Monarch selection.* A processor (called the monarch) can be selected to perform the system initialization process.
- b) *Memory controller.* A standard memory controller can be initialized by firmware, before system software is available.
- c) *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 that shall be provided by a bus standard, and the subset that 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 sometimes data from the requester to the responder. The response subaction returns the status and sometimes data from the responder to the requester. For the read and write transactions, data are only transferred in the response or 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 among node and unit architectures, the CSR Architecture specifies several properties of bus transactions: their address formats (which are described in clause 4), the transaction commands, the data-transfer sizes, and the returned status codes.

3.2 Read and write transactions

Read and write transactions that shall be provided by all bus standards and shall be supported by all memory controllers are specified in table 3. These are aligned transactions; the address shall be an integer multiple of the transaction size.

Table 3—Required transaction types

Read transactions	Write transactions	Size (in bytes)
read1	write1	1
read2	write2	2
read4	write4	4
read8	write8	8
read16	write16	16
read64	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 been 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 4.

Table 4—Optional transaction types

Lock transactions	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 figure 4.

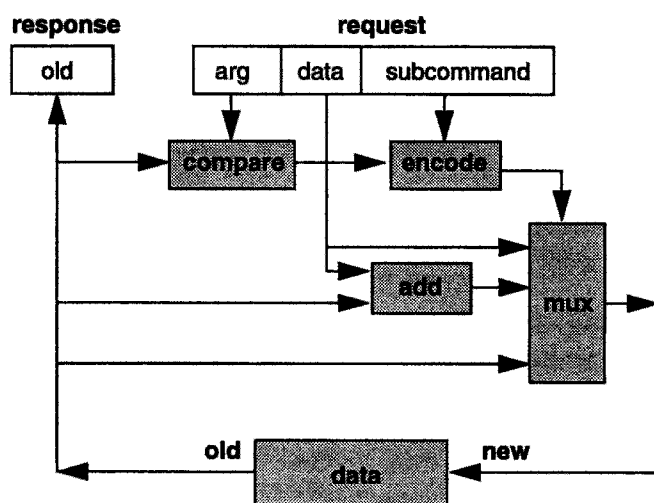


Figure 4—Simplified lock model

The three data values (*data*, *arg*, and *old*) are all the same size and are either 4 or 8 bytes. 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 5. 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.

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.

Table 5—Subcommand values for Lock4 and Lock8

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

* C-code notation used to define update actions.

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

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:

- a) type_error
- b) address_error
- c) conflict_error
- d) 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 to 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:

- a) Some bus standards define a move transaction (a responseless write transaction);
- b) Bridges may buffer write transactions (an optimistic **done_correct** response is returned before the completion status is available); and
- c) 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 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 the 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 figure 5.

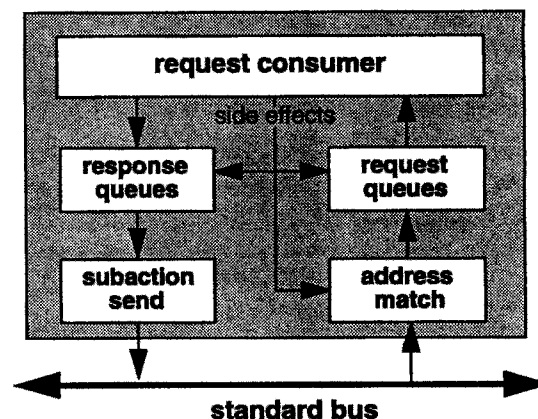


Figure 5—Responder CSR processing model

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-queue entries. These CSR locations are summarized in table 6.

Table 6—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 UNITS_BOUND MEMORY_BASE MEMORY_BOUND	Changes address-match (32/64-bit extended addressing).

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 content 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:

- 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 64 K 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.
- Fixed.* The fixed address-space model supports only 64-bit addresses. The entire 64-bit address space is split into 64 K equal initial node spaces, which are 256 terabytes 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 (64 K) is supported in both address-space mappings, to simplify the address conversions through bus bridges. See 4.2 and 4.3 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 is 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 figure 6.

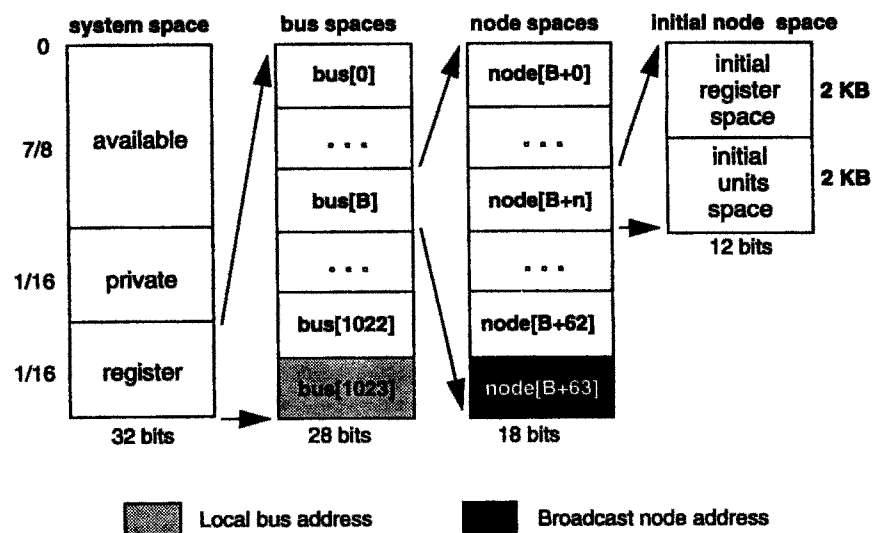


Figure 6—32-bit extended addresses

The 32-bit extended address model supports the available, private, and register spaces, whose address ranges are specified in table 7.

Table 7—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 to 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 figure 7.

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 8. 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.

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).

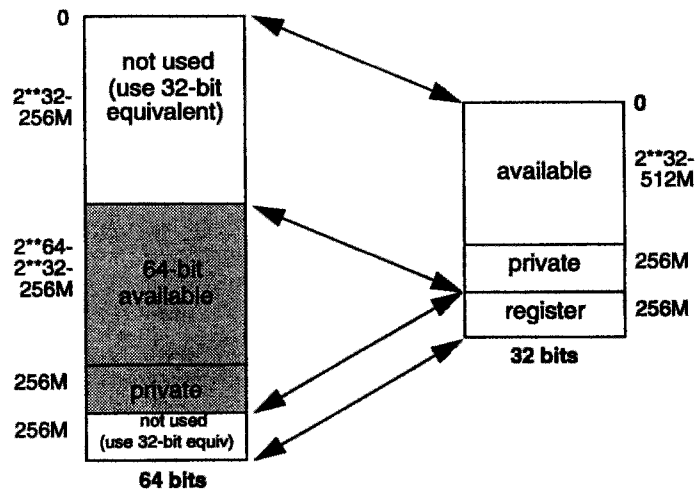


Figure 7—64-bit extended addresses

Table 8—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)

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 64 K 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 terabytes), as illustrated in figure 8.

Up to 64 of the largest *node_id* values are allocated for bus-dependent uses. The shaded blocks illustrate these addresses, which are unavailable 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.

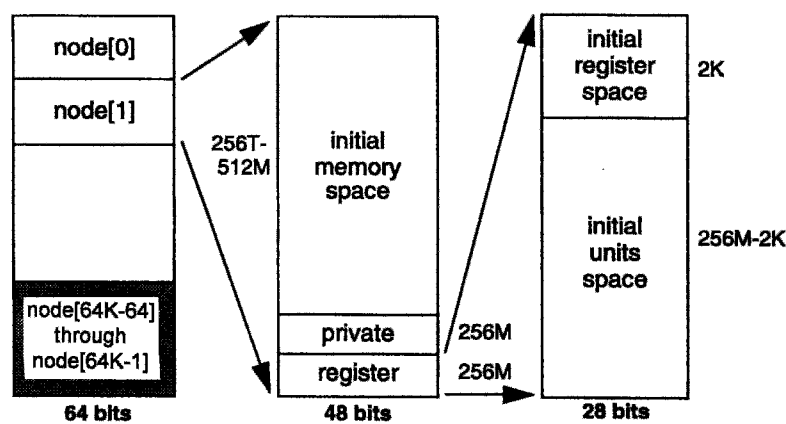


Figure 8—64-bit fixed addressing

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.

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 addressable window to the lower portion of the node's ROM, as illustrated in figure 9.

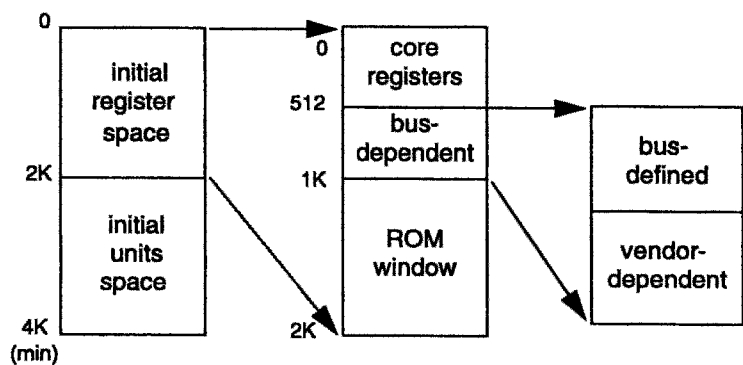


Figure 9—Initial node space components

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 example, a memory 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 figure 10.

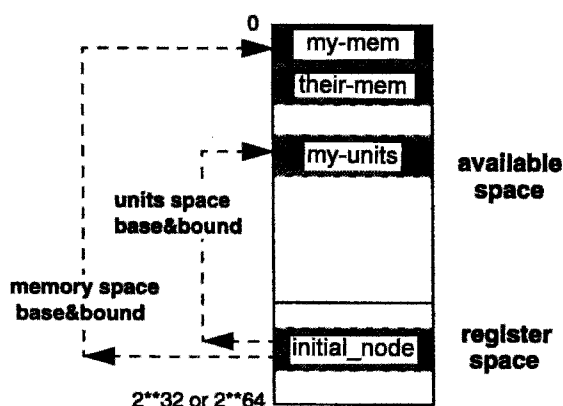


Figure 10—Configurable extended addresses

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 higher 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. 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, for 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 figure 11.

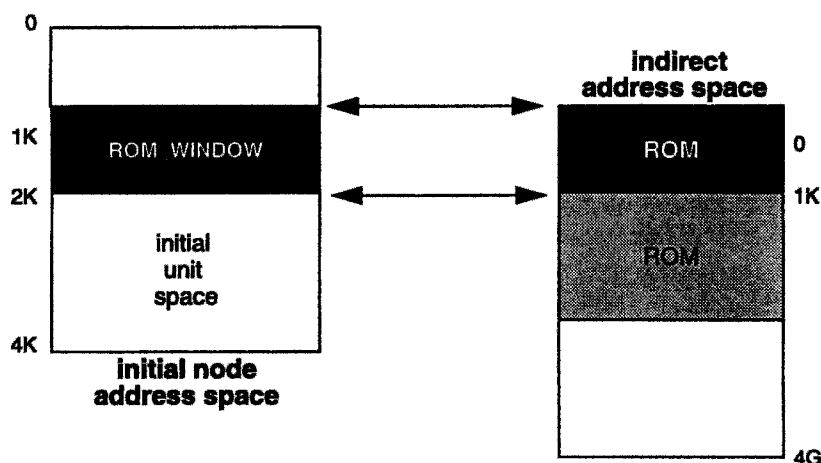


Figure 11—Indirect space mapping (address less than 1 kbyte)

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 prespecified 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 figure 12.

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.